UCID-30179
REVISION 2
COMPUTER
DOCUMENTATION

LAWRENCE LIVERMORE LABORATORY
University of California / Livermore, California

TIMING CODES ON THE CRAY-1: PRINCIPLES AND APPLICATIONS

Harry L. Nelson

May 10, 1981

Prepared for U.S. Department of Energy under Contract No. W-7405-Eng-48

AVAILABILITY

This document is available online as follows:

```
    XPORT RD .717675:UCID:UCID30179 / 1 1
```

View the print file on the TMDS, or print it as follows:

```
    TRIX AC / 1 1
    .PRINT(<NIP UCID30179 BOX ann identification>)
    .END
```

CONTENTS

TIMING CODES ON THE CRAY-1: PRINCIPLES AND APPLICATIONS

ABSTRACT

Complete instruction-timing information for the CRAY-1 computer is presented together with a method of recording the minimum necessary details for precise prediction of the running time of various algorithms. Several examples of optimum assembly language coding are listed, with comments that illustrate the timing details. Usage of the code CYCLES which predicts timing of actual CAL, CFT, or CIVIC programs is described. Usage of codes TIMER and TALLY is described.

I. INTRODUCTION

The aim of this document is to show how to locate and analyze the segments of a code that are important from a timing viewpoint. Computer codes TIMER and TALLY are useful for this purpose. Then, having identified critical sections, we consider how to perform them optimally. Computer code CYCLES is of value in obtaining such performance.

On the CRAY-1, optimum programming consists of finding the best algorithm and avoiding conflicts in implementing it. Usually the best algorithm can be characterized as a "parallel vector" algorithm.

Once an algorithm has been decided upon, one must consider how it can be implemented with actual hardware instructions. The algorithm may have to be changed if it causes unavoidable conflicts due to the shared nature of the CRAY-1's data paths, registers, functional units, and memory. Avoiding conflicts is primarily a matter of understanding the timing details involved.

Several examples of improved performance achieved through timing analysis will be given. (For a description of the environment at LLNL in which your code will run, see Appendix B.)

II. OVERALL TIME ANALYSIS ON THE CRAY-1

The first step in improving the performance of a code is to find out where it is spending its time. In most programs there is some small iterative algorithm that uses the majority of the CPU time. Thus, improvements to a very limited number of lines of code can result in dramatic reductions in the amount of time required to perform a calculation. In particular, if you have a FORTRAN program in which, say, 70% of the time is spent in one inner DO loop, you can limit your effort, initially, to making improvements to that loop. In such cases, obviously, the use of assembly language should be considered. Much of this report will be concerned with time analysis of relatively small assembly language routines. However, initially we look at full code analysis.

Code Timing with TIMER and TALLY

The LASNEX code group, primarily Jim Kohn and George Zimmerman, has put together a simple set of tools to do code timing on the CRAY-1 (and 7600). The capabilities are similar to BEGINMAP-ENDMAP but are simpler to use. The output produced by this set of tools is much less extensive than BEGINMAP but contains the essential ingredients to do timing analysis for almost any code.

Timer

TIMER is a subroutine which you must call in your code. The call looks like:

```
CALL TIMER(IOC,'FNAME',BUFFER,LBUFFER,'HEADER',LHEADER)
```

where,

IOC	is an I/O Connector (IOC) available for I/O. However, if this IOC ever becomes unavailable, TIMER tries to find another one. The IOC is active only during actual writes to disk by TIMER. IOC=0 is satisfactory.
FNAME	is a file sequence name. A sequenced name is formed from this by appending a digit (usually 0) on the right end of the name truncating the leftmost character if necessary. If FNAME already ends with a decimal digit, FNAME is used as is for the first file in the sequence. If any file in the sequence already exists it will be destroyed.
BUFFER	is an I/O buffer. It must be permanently available and reserved for TIMER's use only. Otherwise garbage could be written to disk.

LBUFFER	is the length of the I/O buffer. It may be any size convenient for the user. 512 words seems to work quite well.
HEADER	is an ASCII string which will be written into the beginning of the disk file to identify this timing file (in case multiple runs are made). Date, time, code name, problem name are some possible items that you may wish to put in the header.
LHEADER	is the word length of HEADER. It must be at least one word long, even if the header itself is blank.

TIMER operates by interrupting your code every 4 milliseconds and finding out what the p-counter is. It stores the p-counter in the buffer, dumps the buffer if necessary, and then returns from interrupt. TIMER itself does not perform any actual timing analysis. It just creates a timing file with p-counters in it. The actual analysis is done by the TALLY code.

To obtain a complete timing analysis of your code, TIMER should be called as early as possible during the execution of your code. Once the call to TIMER has been made, no other calls are required until your code wants to terminate the timing analysis. Your code should not be affected by the presence of TIMER in it. The overhead is approximately 5 microseconds per interrupt, which should not be detectable. TIMER contains only about 100 lines of FORTRAN so it is very small.

To terminate the timing analysis, a call must be made to TIMEND. TIMEND is called with no arguments. It shuts down the timing, flushes the buffer, closes the file and truncates it. TIMEND is an entry point inside TIMER.

No externals are required by TIMER (or TIMEND). It is self-contained. It is available by loading your code with ALIBCRAY. If you cannot access ALIBCRAY, the source for TIMER may be extracted from file CLASS, and compiled to produce a binary file for LDR.

TIMER stores one other piece of information in the timing file along with the p-counter. This is a process index. This index is read from common block /Q8LDBKX/ which is one word long. By default this word is set to 1. Your code may set this word at any time to designate the current process which is active. The only reason to do this would be to obtain a more detailed breakdown of the usage of utility subroutines (e.g., SQRT, LOG, EXP, BASELIB routines, etc.) according to the structure of your code. For example, you could find out which logical process in your code is using SQRT the most. This feature is usually used in overlayed (or segmented) codes where the overlay (or segment) number can be stored into this common block. But any single level code could use this equally well. Maximum value for this process index is 255 on CRAY.

Tally

The TALLY code requires 2 files in order to do a timing analysis. The first is the set of timing files (usually 1 file) produced by the TIMER routine. The second file is the symbol table file produced by the loader. The symbol table is usually contained in your controllee file so you may normally use your executing code name as the symbol table file. A copy of TALLY can be extracted from public file "NELSON", at LLNL.

The execute line to run TALLY is:

```
TALLY timing-file-name symbol-table-file [options] / t v
```

where the following options are available,

none	(i.e., no options specified). This does a short timing analysis. Histograms on a subroutine by subroutine basis are not produced.
ALL.	This does a complete timing analysis producing all of the output TALLY can. Most people use this option.
BS. n	Set the Bin Size to n parcels. Tally accumulates timing information into bins. Each bin represents n parcels of your code. Default is n=32 (8 words) which works very nicely.

The timing analysis produced by TALLY is fairly straightforward to understand. It is broken into 3 logical sections. Each sections includes percentage breakdowns as well as actual numbers of hits. The term "hit" designates an instance of the p-counter being in a given routine or a given bin.

The first section does on overall timing analysis. The number of hits in each subprogram as well as the percent of the total time the subprogram used is listed. A subprogram appears in this list only if at least 1 hit was recorded within its bounds.

The second section does a similar kind of analysis but by process index. Thus this is a bit more detailed. The usage of commonly used utility subprograms is broken up by process index.

The third section (if requested with ALL.) is a detailed analysis (via histogram) of each subprogram for which hits were recorded. The breakdown is by bins where a bin represents a small section of code. The number of hits within a bin is printed along with a 'bar' indicating graphically the relative time spent within the bin. Note that the algorithm determining the length of the 'bar' is non-linear. The actual hit count must be used for an accurate, detailed analysis.

Example of output from TALLY.

First, for a GRAFLIB 'typical' test problem written to identify those routines in which time was being spent.

```
01/27/81

NHIT= 998

LOCATION    LENGTH   SUBROUTINE  NHIT PERCENT

00061626 00000635 MAIN.          3   .3006
00063560 00000015 RNFL           3   .3006
00064764 00000515 JPPL2A         1   .1002
00067425 00000106 ZMOVEBIT       7   .7014
00071430 00000634 KXDRPL         2   .2004
00072275 00000040 ZMOVEWRD       3   .3006
00073725 00000146 KXVT2D       126 12.6253
00074073 00000230 KXCL2D       444 44.4890
00075740 00002450 KPFRLN       356 35.6713
00110614 00000070 QBPAK         51  5.1102
00113633 00000041 IZIOSTAT       2   .2004
```

Second, after about one personal month of effort spent recording the three main time-consuming routines into CALL.

```
03/16/81

NHIT=     218

LOCATION    LENGTH   SUBROUTINE  NHIT PERCENT

00061653 00000635 MAIN.          2   .9174
00063605 00000015 RNFL           4  1.8349
00063622 00000121 ZCITOA         1   .4587
00065276 00000626 JPPL2A         1   .4587
00070173 00000106 ZMOVEBIT       4  1.8349
00074424 00000620 KXDRPL         1   .4587
00075255 00000040 ZMOVEWRD       4  1.8349
00077130 00000040 HKXVT2D        7  3.2110
00077170 00000076 HCL2D        128 58.7156
00101335 00002410 KPFRLN         6  2.7523
00114365 00000525 KFRVEC        10  4.5872
00115112 00000070 QBPAK         49 22.4771
00115202 00000060 KWBFFN         1   .4587
```

Another month spent developing and coding vector versions of HCL2D and QBPAK reduced them to 34 and 19 hits respectively, and resulted in a final tenfold improvement for this heavily used LLNL utility, (NHIT= 97).

FLOWTRACE

Often one would like to find out which subroutines of a large code are frequently called and gain an overall knowledge of its flow. CFT users can accomplish this by using FLOWTRACE. This is a compile-time option, which, although expensive, does produce a rather nice breakdown of a code's behavior.

An example of the output from FLOWTRACE is shown below. Full details and assistance are available from the local CRAY representatives.

```
    ROUTINE              TIME        %   CALLED   AVERAGE T

  1 FENBTV            0.059817    1.18        1   0.059817
                                                            CALLS THGEN
  2 THGEN             0.067451    1.33       23   0.002933 CALLED BY FENBTV
  3 BCOND             0.034805    0.68        1   0.034805 CALLED BY FENBTV
  4 ICOND             0.023386    0.46        1   0.023386 CALLED BY FENBTV
  5 PREFRON           0.001754    0.03        1   0.001754 CALLED BY FENBTV
  6 VSTRAP            0.087725    1.72        1   0.087725 CALLED BY FENBTV
                                                            CALLS OUTSOL
  7 OUTSOL            1.190628   23.41       46   0.025883 CALLED BY VSTRAP
  8 FRONT             1.455038   28.60       22   0.066138 CALLED BY VSTRAP
                                                            CALLS QVSET
  9 QVSET             0.010048    0.20       94   0.000107 CALLED BY FRONT
 10 MAKEL             0.035852    0.70        6   0.005975 CALLED BY FRONT
                                                            CALLS QVSET
 11 BASIS             0.000900    0.02        9   0.000100 CALLED BY MAKEL
 12 MAKEQ             0.226627    4.45      132   0.001717 CALLED BY FRONT
                                                            CALLS NLMAT
 13 NLMAT             0.117950    2.32      132   0.000894 CALLED BY MAKEQ
                                                            CALLS ENCOM
 14 ENCOM             0.017083    0.34      132   0.000129 CALLED BY NLMAT
 15 NLRHS             0.001008    0.02        6   0.000168 CALLED BY MAKEQ
 16 BACSUB            0.679884   13.37       22   0.030904 CALLED BY FRONT
 17 ITER              0.058392    1.15       21   0.002781 CALLED BY VSTRAP
                                                            CALLS QVSET

***     TOTAL         5.087028
*** OVERHEAD          0.033296

SUBROUTINE LINKAGE OVERHEAD SUMMARY                 922 CALLS

                MINIMUM     MAXIMUM     AVERAGE     CYCLES    SECONDS         %
T REGISTERS           0          22         6.2      28594   3.57e-04    0.0070
B REGISTERS           2           8         4.3      26306   3.29e-04    0.0065
  ARGUMENTS           0           5         0.8       2876   3.60e-05    0.0007
      total                                          57776   7.22e-04    0.0142
 MAXIMUM SUBROUTINE DEPTH = 7
```

Call Second(0)

Gathering timing information can be made an integral part of a routine. A basic tool I recommend for this use within a specific FORTRAN subroutine is the FORTLIB function SECOND. On the CRAY-1, SECOND returns the total unweighted CPU time charged against your code since execution began. Calls to SECOND are relatively cheap (approximately 5 microseconds per call) and are not subject to variations due to the current time-sharing load on the machine. Other techniques may be used for finer analysis of small code sections, but for overall purposes SECOND is adequate. An example of its use is shown in the code below.

```
      PROGRAM MF301T(UNIT59=TTY)
      COMMON D(1325)
      DIMENSION (1024)
      CALL LINK('UNIT59=TERMINAL//')
      E = SECOND(0)
      TM = SECOND(0)-E
      TT = TM*976.*25.*4.
      T5 = 0
      T2 = 0
      X = .125
      Y = .015625
      A = 15.5
      WRITE(59,58) A,X,Y
58    FORMAT('CHECKING FOR A = ',F7.4,'  X =  ',F7.5,'  Y =  ',F8.6)
      DO 4 K = 1,25
      B = A+X*K
      DO 1 M=1,1325
1     D(M) = B*B-M
      DO 3 J = 1,976
      C = Y*J
      TA = SECOND(0)
      DO 5 I=1,1024
      F(I) = (C-B*D(I))/2.
5     CONTINUE
      TB = SECOND(0)
      T5 = T5+TB-TA-TM
      TA = SECOND(0)
      DO 2 I=1,1024
      IF(F(I).NE.0) GO TO 2
      E = SECOND(0)
      WRITE(59,60) B,C,D(I),E,I,J,K
60    FORMAT('HIT AT',4F9.4,3I5)
2     CONTINUE
      TB = SECOND(0)
      T2 = T2+TB-TA-TM
3     CONTINUE
4     CONTINUE
```

```
      E = SECOND(0)
      WRITE(59,59) A,E,I,J,K
      WRITE(59,61) T5,T2,TT
61    FORMAT('LOOP5 TIME =',F9.4,3X,'LOOP2 TIME =',F9.4,3X
     % ,'CLOCK CALL TIME =',F9.4)
      STOP 1
59    FORMAT('             A          TIME       I     J     K',/,'NO HIT',
     % 2F9.4,3I5)
      END
```

```
      -------------------------------------------------------------
      !                                                           !
      !Note:  The source code for this example, MF301T, as well as the!
      !sources for all other examples in this writeup are resident on !
      !the CRAY-1 in public LIB file CLASS.  One can extract and run  !
      !this example using the CIVIC compiler as follows (lower case   !
      !typing represents user input; upper case is computer output):  !
      -------------------------------------------------------------
```

```
lib class
C 06/13/79 09:41:03 644400
OK. x mf301t
OK. end

 ALL DONE
civic mf301t mfc
*** CRAY LOADER VERSION - C120  03/08/79

 ALL DONE
mfc
CHECKING FOR A =   15.5000    X =  0.12500    Y =  0.015625
HIT AT   15.7500   0.9844   0.0625    1.6408  248   63     2
HIT AT   16.2500   1.0156   0.0625    7.8015  264   65     6
HIT AT   16.5000   4.1250   0.2500   11.1936  272  264     8
HIT AT   16.7500   9.4219   0.5625   14.8103  280  603    10
HIT AT   17.2500   9.7031   0.5625   20.9958  297  621    14
HIT AT   17.5000   4.3750   0.2500   23.5364  306  280    16
HIT AT   17.7500   1.1094   0.0625   26.2874  315   71    18
HIT AT   18.2500   1.1406   0.0625   32.4474  333   73    22
HIT AT   18.5000   4.6250   0.2500   35.8796  342  296    24
             A       TIME       J      J     K
NO HIT   15.500   38.4952  1025   977    26
LOOP5 TIME =   15.6641   LOOP2 TIME =   18.4826   CLOCK CALL TIME =   4.2944
```

The following, for comparison, is the CFT version, which is automatically vectorized for loop 5:

```
  rcft i=mf301t,go
  CF000 - CFT VERSION -   01/23/81 1.09b
  CF001 - COMPILE TIME =    0.0346 SECONDS
  CF002 -        54 LINES,        44 STATEMENTS
  *** CRAY LOADER VERSION - C120  03/08/79
  CHECKING FOR A =  15.5000    X =  0.12500    Y =  0.015625
  HIT AT  15.7500   0.9844   0.0625   0.9592  248   63    2
  HIT AT  16.2500   1.0156   0.0625   4.5498  264   65    6
  HIT AT  16.5000   4.1250   0.2500   6.5256  272  264    8
  HIT AT  16.7500   9.4219   0.5625   8.6349  280  603   10
  HIT AT  17.2500   9.7031   0.5625  12.2467  297  621   14
  HIT AT  17.5000   4.3750   0.2500  13.7301  306  280   16
  HIT AT  17.7500   1.1094   0.0625  15.3368  315   71   18
  HIT AT  18.2500   1.1406   0.0625  19.9301  333   73   22
  HIT AT  18.5000   4.6250   0.2500  20.9280  342  296   24
            A        TIME      I     J     K
  NO HIT  15.5000  22.4518  1025   977   26
  LOOP5 TIME =   1.1869   LOOP2 TIME =  16.9523   CLOCK CALL TIME =  4.1968
```

From these numbers, we can see that (for the CFT version, at least) improvement efforts should be directed toward loop 2. (And, of course, the calls to SECOND will be eventually removed.)

IRTC and/or Q8RTC

The CRAY-1 has a cycle counter as one of its hardware features. This is a counter which steps by one each machine clock period of 12.5 nanoseconds. Detailed timing of code sections can be done using this counter. However, the counter steps whether or not your program is running, so care must be taken with its use in the time-sharing environment. The counter, called RTC (for real-time clock), is directly readable using FORTRAN. With CFT, one uses the construct. N = IRTC(0), and with CIVIC, N = Q8RTC(0), where N is an integer variable name. The compiler generates only the code necessary for reading the RTC and storing the reading in memory location N, a total of 48 bits of code, normally requiring only 3 extra clock periods to perform. (In certain cases a longer time is required because of an S-register, path, or memory conflict.)

The use of IRTC is illustrated in the session below. In the example, a FORTRAN routine calls a CAL assembly routine, which adds the first 51 elements of arrays A and B and places the result into array C by use of a scalar loop.

Here, it was possible to improve the performance of the machine on this example by about 6% by merely reordering the modules in memory. There are (admittedly pathological) examples of this type of thing where a change in running time of 100% occurs. Such changes are due to the avoidance of (or introduction of) conflicts.

First, the source codes for the example are extracted.

```
lib class
C 07/06/79 13:19:54 644400
OK. x abcs abcsf
OK. end
 ALL DONE
```

```
trixgl o!abcs
     19 LINES (   80S)
.t
 1 *       CAL  I=ABCS,B=BABCS,L=LSC
 2            IDENT          ABCS
 3            COMMON       ABCOMMON
 4 A          BSS            57
 5 B          BSS            56
 6 C          BSS            56
 7            BLOCK          ABCS
 8            ENTRY          ABCS
 9 ABCS         A1             0
10            A2             51
11  LOOP      S1             A,A1
12            S2             B,A1
13            S3             S1+FS2
14            C,A1           S3
15            A1             A1+1
16            A0             A1-A2
17            JAN            LOOP
18            J              B00
19            END
.run
CAL  I=ABCS,B=BABCS,L=LSC
%PC3
[3.000]
CA012 - 0062K MEMORY + 0117K I/O BUFFERS USED

 ALL DONE


o!abcsf
    17 LINES (   80S)
.t
 1 *      CFT  I=ABCSF,ON=G,L=LSF,B=BSF
 2 *        LDR  I=(BSF,BABCS),ML=MSF,X=XBS,ORDER=CLNB,FIRST=BSF
 3 *      XBS
 4          COMMON /ABCOMMON/ A(56),OUTRANGE,B(56),C(56)
 5          CALL LINK('UNIT59=TERMINAL//')
 6          Y = X*X*X*X*X*X*Y*X
 7          DO 1 I = 1,169
 8    1     A(I) = I
 9          OUTRANGE = 600004000000000000000B
10          M = IRTC(0)
11          CALL ABCS
12          N = IRTC(0)
13          X = N-M
14          WRITE(59,59) C,X
15   59     FORMAT(7F6.0)
16          STOP
17          END
```

```
.run
CFT I=ABCSF,ON=G,L=LSF,B=BSF
FT004 - CFT VERSION -     04/06/79 SCHEDULER
FT001 - COMPILE TIME =         0.0195 SECONDS

 ALL DONE
LDR I=(BSF,BABCS),ML=MSF,X=XBS,ORDER=CLNB,FIRST=BSF

 ALL DONE
XBS
    59.    61.    63.    65.    67.    69.    71.
    73.    75.    77.    79.    81.    83.    85.
    87.    89.    91.    93.    95.    97.    99.
   101.   103.   105.   107.   109.   111.   113.
   115.   117.   119.   121.   123.   125.   127.
   129.   131.   133.   135.   137.   139.   141.
   143.   145.   147.   149.   151.   153.   155.
   157.   159.   165.   166.   167.   168.   169.
  1773.
STOP
```

The last number listed (1773) is the number of machine cycles elapsing between the two uses of IRTC in the code ABCSF.

Notice, next, the result of an apparently innocuous change to line 2.

```
rp2!=BSF!=BABCS

.nf!run
     17 LINES (   80S)
CFT I=ABCSF,ON=G,L=L=LSF,B=FSF
FT004 - CFT VERSION -      04/06/79 SCHEDULER
FT001 - COMPILE TIME =       0.0191 SECONDS

 ALL DONE
LDR I=(BSF,BABCS),ML=MSF,X=XBS,ORDER=CLNB,FIRST=BABCS

 ALL DONE
XBS
    59.    61.    63.    65.    67.    69.    71.
    73.    75.    77.    79.    81.    83.    85.
    87.    89.    91.    93.    95.    97.    99.
   101.   103.   105.   107.   109.   111.   113.
   115.   117.   119.   121.   123.   125.   127.
   129.   131.   133.   135.   137.   139.   141.
   143.   145.   147.   149.   151.   153.   155.
   157.   159.   165.   166.   167.   168.   169.
  1659.
STOP

 ALL DONE
```

Øther Methods

Øne can use the 072 machine instruction directly to discover ultra-fine timing details related to hardware and special code loops. This detail is made available to the CRAY-1 programmer through use of the public file "CYCLES". See Section IV for more information.

III. PREDICTING TIMING

The rest of this paper will be used to demonstrate (and, I hope, teach you) a method for explicitly predicting timing. The method can help in avoiding unnecessary conflicts in assembly-language-coded subroutines or in loops which one expects to utilize considerable machine time and for which, therefore, one is justified in spending considerable human time to obtain top performance. Since the method outlined is almost completely mechanical, a program using these ideas has been written to generate timing charts such as those shown below. The program is called CYCLES. Its usage is described in Section IV of this report.

I will assume that the reader is familiar with the CRAY-1 Hardware Manual and CAL assembly language. In particular, the five pages of our Appendix A, taken from the CRAY-1 Hardware Manual, list much of the information needed for timing purposes. Examples will be either given in CAL or, on occasion, taken directly from the long listing of CFT or CIVIC.

General Remarks

In general, the time required to perform an algorithm depends on the specific instructions used to perform it and on the relationships among those instructions. A complete understanding of the relevant conditions affecting the execution of a particular instruction can be gained only by considering its relation to surrounding instructions. In particular, vector instructions require somewhat more analysis than scalars.

I find that recording at most five easily computed numbers per instruction will give the necessary information for determining conflicts and suggesting ways to avoid them. For a scalar (or register) instruction one needs to keep track of: (1) when it issues, and (2) when it completes. For a vector instruction one has to note: (1) its issue time, (2) its chain time, and the (different) times when it has finished using: (3) its input registers, (4) its functional unit, and (5) its output register.

In all cases, except for scalar memory-referencing instructions (and normally it is true then, also), once the issue cycle has been determined, all the other timing numbers for that instruction are computable. The rules for doing these computations are stated on page 25 of this report, and the exceptions are noted in appropriate examples.

Table 1 (adapted from Appendix D of the CRAY-1 Hardware Manual) lists the entire set of timing numbers (first column) needed for most purposes. These specify the number of 12.5 nanosecond machine cycles required by the CRAY-1 to deliver a result to the appropriate register. (0 means no result goes to a register.) Further detail is available in Chapter 4 of the Cray-1 Hardware Manual in conjunction with each specific instruction description.

Note. All instructions using the Memory Functional Unit are subject to possible additional delays due to memory bank conflicts with I/O.

Table 1. Instruction and Timing Summary

Cy-cles	CRAY-1	CAL mnemonics		Unit	Description
∞	000xxx	ERR		-	Error exit
50	**000ijk	ERR	exp	-	Error exit
0	*001000	NOP		-	No operation
1	**0010jk	CA,Aj	Ak	-	Set the channel (Aj) current address to (Ak) and begin the I/O sequence
1	**0011jk	CL,Aj	Ak	-	Set the channel (Aj) limit address to (Ak)
1	**0012jx	CI,Aj		-	Clear channel (Aj) interrupt flag
1	**0013jx	XA	Aj	-	Enter XA register with (Aj)
1	**0014jx	RT	Sj	-	Enter real-time clock register with (Sj)
1	**0014j4	PCI	Sj	-	Enter II with (Sj)
1	**0014j5	CCI		-	Clear clock interrupt
1	**0014j6	ECI		-	Enable Clock interrupt
1	**0014j7	DCI		-	Disable clock interrupt
1	0020xk	VL	Ak	-	Transmit (Ak) to VL register
1	*0020x0	VL	1	-	Transmit 1 to VL register
1	0021xx	EFI		-	Enable interrupt on flt pt error
1	0022xx	DFI		-	Disable interrupt on flt pt error
3	003xjx	VM	Sj	-	Transmit (Sj) to VM register
3	*003x0x	VM	0	-	Clear VM register
∞	004xxx	EX		-	Normal exit
50	**004ijk	EX		-	Normal exit
7(+)	005xjkx	J	Bjk	-	Jump to (Bjk)
5(+)	006ijkm	J	exp	-	Jump to exp
5(+)	007ijkm	R	exp	-	Return jump to exp; set B00 to P
5(+)	010ijkm	JAZ	exp	-	Branch to exp if (A0) = 0
5(+)	011ijkm	JAN	exp	-	Branch to exp if (A0).NE.0
5(+)	012ijkm	JAP	exp	-	Branch to exp if (A0) positive
5(+)	013ijkm	JAM	exp	-	Branch to exp if (A0) negative
5(+)	014ijkm	JSZ	exp	-	Branch to exp if (S0) = 0
5(+)	015ijkm	JSN	exp	-	Branch to exp if (S0).NE.0
5(+)	016ijkm	JSP	exp	-	Branch to exp if (S0) positive
5(+)	017ijkm	JSM	exp	-	Branch to exp if (S0) negative
1	020ijkm			-	Transmit exp = jkm to Ai
1	021ijkm	Ai	exp	-	Transmit exp = 1's complement of jkm to Ai
1	022ijk	Ai	exp	-	Transmit exp = jk to Ai
1	023ijx	Ai	Sj	-	Transmit (Sj) to Ai
1	024ijk	Ai	Bjk	-	Transmit (Bjk) to Ai

----

* Special CAL syntax form.
** Privileged to monitor mode.
x Indicates that the field is not used by the hardware; the assembler generates a zero in this position.
+ These jump instructions take longer if branched-to address is not already in an instruction buffer. They then use the memory functional unit.

Cy-cles	CRAY-1	CAL mnemonics	Unit	Description
1	025ijk	Bjk Ai	-	Transmit (Ai) to Bjk
4	026ix0	Ai PSj	Pop/LZ	Population count of (Sj) to Ai
4	026ij1	Ai QSj	Pop/LZ	Pop count parity of (Sj) to Ai
3	027ijx	Ai ZSj	Pop/LZ	Leading zero count of (Sj) to Ai
2	030ijk	Ai Aj+Ak	A Int Add	Integer sum of (Aj) and (Ak) to Ai
2	*030i0k	Ai Ak	A Int Add	Transmit (Ak) to Ai
2	*030ij0	Ai Aj+1	A Int Add	Integer sum of (Aj) and 1 to Ai
2	031ijk	Ai Aj-Ak	A Int Add	Integer difference of (Aj) less (Ak) to Ai
2	*031i00	Ai -1	A Int Add	Transmit -1 to Ai
2	*031i0k	Ai -Ak	A Int Add	Transmit the negative of (Ak) to Ai
2	*031ij0	Ai Aj-1	A Int Add	Integer difference of (Aj) less 1 to Aj
6	032ijk	Ai Aj*Ak	A Int Mult	Integer product of (Aj) and (Ak) to Ai
4	*033i0x	Ai CI	-	Channel number to Ai (j=0)
4	*033ij0	Ai CA,Aj	-	Address of channel (Aj) to Ai (j.NE.0)
4	033ij1	Ai CE,Aj	-	Error flag of channel (Aj) to Ai (j.NE.0)
14(+)	034ijk	Bjk,Ai ,A0	Memory	Read (Ai) words to B register jk from (A0)
14(+)	*034ijk	Bjk,Ai 0,A0	Memory	Read (Ai) words to B register jk from (A0)
6(+)	035ijk	,A0 Bjk,Ai	Memory	Store (Ai) words at B register jk to (A0)
6(+)	*035ijk	0,A0 Bjk,Ai	Memory	Store (Ai) words at Be register jk to (A0)
14(+)	036ijk	Tjk,Ai ,A0	Memory	Read (Ai) words to T register jk from (A0)
14(+)	*036ijk	Tjk,Ai 0,A0	Memory	Read (Ai) words to T register jk from (A0)
6(+)	037ijk	,A0 Tjk,Ai	Memory	Store (Ai) words at T register jk to (A0)
6(+)	*037ijk	0,A0 Tjk,Ai	Memory	Store (Ai) words at T register jk to (A0)
1	040ijkm	Si exp	-	Transmit jkm to Si
1	041ijkm	Si exp	-	Transmit exp = 1's complement of jkm to Si
1	042ijk	Si <exp	S Logical	Form 1's mask exp = 64-jk bits in Si from the right
1	*042ijk	Si #>exp	S Logical	Form 0's mask exp = jk bits in Si from the left
1	*042i00	Si -1	S Logical	Enter -1 into Si

----
* Special CAL syntax form.
+ The cycles needed = this number + (Ai). Also, no issues allowed till completion.
x Field not used.

Cy-cles	CRAY-1	CAL mnemonics	Unit	Description
1	*042i77	Si 1	S Logical	Enter 1 into Si
1	043ijk	Si >exp	S Logical	Form 1's mask exp = jk bits in Si from the left
1	*043ijk	Si #<exp	S Logical	Form 0's mask exp = 64-jk bits in Si from the right
1	*043i00	Si 0	S Logical	Clear Si
1	044ijk	Si Sj&Sk	S Logical	Logical product of (Sj) and (Sk) to Si
1	*044ij0	Si Sj&SB	S Logical	Sign bit of (Sj) to Si
1	*045ijk	Si #Sk&Sj	S Logical	Logical product of (Sj) and 1's complement of (Sk) to Si
1	*045ij0	Si #SB&Sj	S Logical	(Sj) with sign bit cleared to Si
1	046ijk	Si Sj\Sk	S Logical	Logical difference of (Sj) and (Sk) to Si
1	*046ij0	Si Sj\SB	S Logical	Toggle sign bit of Sj, then enter into Si
1	*046ij0	Si SB\Sj	S Logical	Toggle sign bit of Sj, then enter into Si (j.NE.0)
1	047ijk	Si #Sj\Sk	S Logical	Logical equivalence of (Sk) and (Sj) to Si
1	*047i0k	Si #Sk	S Logical	Transmit 1's complement of (Sk) to Si
1	*047ij0	Si #Sj\SB	S Logical	Logical equivalence of (Sj) and sign bit to Si
1	*047i00	Si #SB	S Logical	Enter 1's complement of sign bit into Si
1	050ijk	Si Sj!Si&Sk	S Logical	Logical product of (Si) and (Sk) complement ORed with logical product of (Sj) and (Sk) to Si
1	*050ij0	Si Sj!Si&SB	S Logical	Scalar merge of (Si) and sign bit of (Sj) to Si
1	051ijk	Si Sj!Sk	S Logical	Logical sum of (Sj) and (Sk) to Si
1	*051i0k	Si Sk	S Logical	Transmit (Sk) to Si
1	*051ij0	Si Sj!SB	S Logical	Logical sum of (Sj) and sign bit to Si
1	*051i00	Si SB	S Logical	Enter sign bit into Si
2	052ijk	S0 Si<exp	S Shift	Shift (Si) left exp = jk places to S0
2	053ijk	S0 Si>exp	S Shift	Shift (Si) right exp = 64-jk places to S0
2	054ijk	Si Si<exp	S Shift	Shift (Si) left exp = jk places
2	055ijk	Si Si>exp	S Shift	Shift (Si) right exp = 64-jk places
3	056ijk	Si Si,Sj<Ak	S Shift	Shift (Si and Sj) left (Ak) places to Si
3	*056ij0	Si Si,Sj<1	S Shift	Shift (Si and Sj) left one place to Si

----

* Special CAL syntax form.

Cy-cles	CRAY-1	CAL mnemonics		Unit	Description
3	*056i0k	Si	Si<Ak	S Shift	Shift (Si) left (Ak) places to Si
3	057ijk	Si	Sj,Si>Ak	S Shift	Shift (Sj and Si) right (Ak) places to Si
3	*057ij0	Si	Sj,Si>1	S Shift	Shift (Sj and Si) right one place o Si
3	*057i0k	Si	Si>Ak	S Shift	Shift (Si) right (Ak) places to Si
3	060ijk	Si	Sj+Sk	S Int Add	Integer sum of (Sj) and (Sk) to Si
3	061ijk	Si	Sj-Sk	S Int Add	Integer difference of (Sj) and (Sk) to Si
3	*061i0k	Si	-Sk	S Int Add	Transmit negative of (Sk) to Si
6	062ijk	Si	Sj+FSk	F.P. Add	Floating sum of (Sj) and (Sk) to Si
6	*062i0k	Si	+FSk	F.P. Add	Normalize (Sk) to Si
6	063ijk	Si	Sj-FSk	F.P. Add	Floating difference of (Sj) and (Sk) to Si
6	*063i0k	Si	-FSK	F.P. Add	Transmit normalized negative of (Sk) to Si
7	064ijk	Si	Sj*FSk	F.P. Mult	Floating product of (Sj) and (Sk) to Si
7	065ijk	Si	Sj*HSk	F.P. Mult	Half precision rounded floating product of (Sj) and (Sk) to Si
7	066ijk	Si	Sj*RSk	F.P. Mult	Full precision rounded floating product of (Sj) and (Sk) to Si
7	067ijk	Si	Sj*ISk	F.P. Mult	2 - Floating product of (Sj) and (Sk) to Si
14	070ijx	Si	/HSj	F.P. Rcpl	Floating reciprocal approximation of (Sj) to Si
2	071i0k	Si	Ak	-	Transmit (Ak) to Si with no sign extension
2	071i1k	Si	+Ak	-	Transmit (Ak) to Si with sign extension
2	071i2k	Si	+FAk	-	Transmit (Ak) to Si as unnormalized floating point number
2	071i3x	Si	0.6	-	Transmit constant 0.75*2**48 to Si
2	071i4x	Si	0.4	-	Transmit constant 0.5 to Si
2	071i5x	Si	1.	-	Transmit constant 1.0 to Si
2	071i6x	Si	2.	-	Transmit constant 2.0 to Si
2	071i7x	Si	4.	-	Transmit constant 4.0 to Si
1	072ixx	Si	RT	-	Transmit (RTC) to Si
1	073ixx	Si	VM	-	Transmit (VM) to Si
1	074ijk	Si	Tjk	-	Transmit (Tjk) to Si
1	075ijk	Tjk	Si	-	Transmit (Si) to Tjk

----

* Special CAL syntax form.
x Field not used.

Cy-cles	CRAY-1	CAL mnemonics		Unit	Description
5	076ijk	Si	Vj,Ak	-	Transmit (Vj, element (Ak)) to Si
1	077ijk	Vi,Ak	Sj	-	Transmit (Sj) to Vi element (Ak)
1	*077i0k	Vi,Ak	0	-	Clear Vi element (Ak)
11	10hijkm	Ai	exp,Ah	Memory	Read from ((Ah) + exp) to Ai (A0=0)
11	*100ijkm	Ai	exp,0	Memory	Read from (exp) to Ai
11	*100ijkm	Ai	exp,	Memory	Read from (exp) to Ai
11	*10hi000	Ai	,Ah	Memory	Read from (Ah) to Ai
0	11hijkm	exp,Ah	Ai	Memory	Store (Ai) to (Ah) + exp (A0=0)
0	*110ijkm	exp,0	Ai	Memory	Store (Ai) to exp
0	*110ijkm	exp,	Ai	Memory	Store (Ai) to exp
0	*11hi000	,Ah	Ai	Memory	Store (Ai) to (Ah)
11	12hijkm	Si	exp,Ah	Memory	Read from ((Ah) + exp) to Si (A0=0)
11	*120ijkm	Si	exp,0	Memory	Read from exp to Si
11	*120ijkm	Si	exp,	Memory	Read from exp to Si
11	*12hi000	Si	,Ah	Memory	Read from (Ah) to Si
0	13hijkm	exp,Ah	Si	Memory	Store (Si) to (Ah) + exp (A0=0)
0	*130ijkm	exp,0	Si	Memory	Store (Si) to exp
0	*130ijkm	exp,	Si	Memory	Store (Si) to exp
0	*13hi000	,Ah	Si	Memory	Store (Si) to (Ah)
4	140ijk	Vi	Sj&Vk	V Logical	Logical products of (Sj) and (Vk) to Vi
4	141ijk	Vi	Vj&Vk	V Logical	Logical products of (Vj) and (Vk) to Vi
4	142ijk	Vi	Sj!Vk	V Logical	Logical sums of (Sj) and (Vk) to Vi
4	*142i0k	Vi	Vk	V Logical	Transmit (Vk) to Vi
4	143ijk	Vi	Vj!Vk	V Logical	Logical sums of (Vj) and (Vk) to Vi
4	144ijk	Vi	Sj\Vk	V Logical	differences of (Sj) and (Vk) to Vi
4	*145iii	Vi	0	V Logical	Clear Vi
4	145ijk	Vi	Vj\Vk	V Logical	Logical differences of (Vj) and (Vk) to Vi
4	146ijk	Vi	Sj!Vk&VM	V Logical	Transmit (Sj) if VM bit = 1; (Vk) if VM bit = 0 to Vi
4	*146i0k	Vi	#VM&Vk	V Logical	Vector merge of (Vk) and 0 to Vi
4	147ijk	Vi	Vj!Vk&VM	V Logical	Transmit (Vj) if VM bit = 1; (Vk) if VM bit = 0 to Vi
6	150ijk	Vi	Vj<Ak	V Shift	Shift (Vj) left (Ak) places to Vi
6	*150ij0	Vi	Vj<1	V Shift	Shift (Vj) left one place to Vi
6	151ijk	Vi	Vj>Ak	V Shift	Shift (Vj) right (Ak) places to Vi
6	*151ij0	Vi	Vj>1	V Shift	Shift (Vj) right one place to Vi
6	152ijk	Vi	Vj,Vj<Ak	V Shift	Double shift (Vj) left (Ak) places to Vi

----

* Special CAL syntax form.

Cy-cles	CRAY-1	CAL mnemonics		Unit	Description
6	*152ij0	Vi	Vj,Vj<1	V Shift	Double shift (Vj) left one place to Vi
6	153ijk	Vi	Vj,Vj>Ak	V Shift	Double shift (Vj) right (Ak) places to Vi
6	*153ij0	Vi	Vj,Vj>1	V Shift	Double shift (Vi) right one place to Vi
5	154ijk	Vi	Sj+Vk	V Int Add	Integer sums of (Sj) and (Vk) to Vi
5	155ijk	Vi	Vj+Vk	V Int Add	Integer sums of (Vj) and (Vk) to Vi
5	156ijk	Vi	Sj-Vk	V Int Add	Integer differences of (Sj) and (Vk) to Vi
5	*156i0k	Vi	-Vk	V Int Add	Transmit negative of (Vk) to Vi
5	157ijk	Vi	Vj-Vk	V Int Add	Integer differences of (Vj) and (Vk) to Vi
9	160ijk	Vi	Sj*FVk	F.P. Mult	Floating products of (Sj) and (Vk) to Vi
9	161ijk	Vi	Vj*FVk	F.P. Mult	Floating products of (Vj) and (Vk) to Vi
9	162ijk	Vi	Sj*HVk	F.P. Mult	Half precision rounded floating products of (Sj) and (Vk) to Vi
9	163ijk	Vi	Vj*HVk	F.P. Mult	Half precision rounded floating products of (Vj) and (Vk) to Vi
9	164ijk	Vi	Sj*RVk	F.P. Mult	Rounded floating products of (Sj) and (Vk) to Vi
9	165ijk	Vi	Vj*RVk	F.P. Mult	Rounded floating products of (Vj) and (Vk) to Vi
9	166ijk	Vi	Sj*IVk	F.P. Mult	2 - floating products of (Sj) and (Vk) to Vi
9	167ijk	Vi	Vj*IVk	F.P. Mult	2 - floating products of (Vj) and (Vk) to Vi
8	170ijk	Vi	Sj+FVk	F.P. Add	Floating sums of (Sj) and (Vk) to Vi
8	*170i0k	Vi	+FVk	F.P. Add	Normalize (Vk) to Vi
8	171ijk	Vi	Vj+FVk	F.P. Add	Floating sums of (Vj) and (Vk) to Vi
8	172ijk	Vi	Sj-FVk	F.P. Add	Floating differences of (Sj) and (Vk) to Vi
8	*172i0k	Vi	-FVk	F.P. Add	Transmit normalized negatives of (Vk) to Vi
8	173ijk	Vi	Vj-FVk	F.P. Add	Floating differences of (Vj) and (Vk) to Vi
16	174ij0	Vi	/HVj	F.P. Rcpl	Floating reciprocal approximations of (Vj) to Vi
8	174ij1	Vi	PVj	F.P. Rcpl	Population counts of (Vj) to Vi
8	174ij2	Vi	QVj	F.P. Rcpl	Pop count parity of (Vj) to Vi

* Special CAL syntax form.

Cy-cles	CRAY-1	CAL mnemonics		Unit	Description
6	175xj0	VM	Vj,Z	V Logical	VM=1 where (Vj) = 0
6	175xj1	VM	Vj,N	V Logical	VM=1 where (Vj).NE.0
6	175xj2	VM	Vj,P	V Logical	VM=1 where (Vj) positive
6	175xj3	VM	Vj,M	V Logical	VM=1 where (Vj) negative
9	176ixk	Vi	,A0,Ak	Memory	Read (VL) words to Vi from (A0) incremented by (Ak)
9	*176ix0	Vi	,A0,1	Memory	Read (VL) words to Vi from (A0) incremented by 1
0	177xjk	,A0,Ak	Vj	Memory	Store (VL) words from Vj to (A0) incremented by (Ak)
0	*177xj0	,A0,1	Vj	Memory	Store (VL) words from Vj to (A0) incremented by 1

----

* Special CAL syntax form.
x Field not used.

The Basic Details

In general we have the following scenario: in order to perform some alteration of the contents of one or more of the machine's registers or memory, an instruction must: first, wait to be brought into one of the instruction buffers; second, wait until prior instructions have started; third, wait till its operands are available; and fourth, wait until all shared components (such as paths along which information may flow, registers that may be needed, and functional units that may be employed) will be available during the cycle(s) required. The CRAY-1 hardware maintains reservation tables, updated each cycle, for each register and all other shared components. It releases or issues an instruction only when it can be completed without interference from other previously issued instructions.

Generally, timing analysis begins when the first instruction of interest issues, but it is naive not to consider its placement in an instruction buffer and the route by which it reached issuable condition. For many algorithms, speed changes on the order of 10% occur depending on their placement relative to the start of an instruction buffer. Details about the instruction fetch mechanism are found in Appendix C.

All of the information used to decide about the issue of an instruction is contained in its 16 bits or, in the case of a 32-bit instruction, in its upper 16 bits. Normally the decision to issue can be made in one cycle. When an instruction issues, the components it will use are reserved in the appropriate table for the appropriate time period.

One type of 32-bit instruction, which makes a scalar memory reference, is allowed to issue when all of the components it will need are available except possibly the appropriate memory bank. If the bank is available at the proper time, all proceeds normally. If not, completion of the instruction is delayed and the next instruction requesting memory is not allowed to issue until the previous one has obtained the proper memory access. Instructions not requiring memory, however, may proceed normally.

Until a specific instruction issues, the machine cannot look beyond it to determine that something further down in the instruction sequence could be done. It is the task of the programmer and compiler to so order the computation that unnecessary delays are avoided. When you program in assembly language, it is important (and not difficult) to maintain an understanding of the resources of the machine called into play by each instruction and of the cycles in which they are used, in order to approach optimum utilization of the hardware.

During the issue cycle, paths are opened so that information can flow from registers to functional units; during the completion cycle, paths are required for information to flow from functional units to registers. Only one path is available to service all results being returned to any of the eight S-registers. There is also one path for the A-registers. Possible conflicts over the use of these paths are resolved before an instruction is allowed to issue. A separate path into and out of each vector register is provided. Moreover, information arriving at any register in a given cycle may also be redirected by a subsequent instruction, in that same cycle, to serve as input for another operation. That is, a subsequent instruction may issue on the same cycle in which its operands first become available. This redirection of information arriving at a vector register is called chaining, and it may begin only during the particular cycle when the first element of the result is returned from a functional unit. If two different functional units return their first results in the same cycle, a third instruction may chain from both of them.

An exception to this "same cycle rule" occurs for conditional branch instructions, which require that their operand register becomes available somewhat before issue.

Two Short Examples

Let us consider what the hardware must take into account to decide when to issue a couple of typical instructions.

First, a scalar floating point add: 62312, S3 S1+FS2.

When the instruction sequence reaches such an instruction, the hardware checks its reservation tables to see that none of the following conditions are true: (1) the floating point add functional unit is busy (i.e., reserved) in this cycle, (2) register S3 is busy, (3) register S1 is busy, (4) register S2 is busy, (5) a reservation exists for the S-register input path 6 cycles hence. If any of these conditions are true, the instruction does not issue. In the next cycle (the machine having updated all its tables), the same conditions are tested. Eventually, all the needed components will be free and the instruction will issue. When it does, the tables will have: (1) a busy condition placed on S3 for 6 cycles (i.e., cycles 0,1,2,3,4, and 5) and (2) a reservation placed on the S-register input path 6 cycles hence (cycle 6). (No reservation is put on a functional unit by a scalar instruction.) In the next cycle, the next instruction will be considered for issue, and the components it needs will be checked for availability.

Now consider a vector instruction: 171312, V3 V1+FV2.

When this floating point vector add is reached, the hardware checks its reservation tables for the following conditions: (1) floating point adder reserved, (2) vector register V3 busy, (3) V1 busy, and (4) V2 busy. It does not need to check for path reservations since each V-register has its own path. When none of these conditions are true, the instruction issues. When it does, (1) the tables have a busy condition placed on V1 and V2 for, max((VL),5) cycles, where (VL) is the current value of the vector length register (thus for short vectors a minimum reservation of 5 cycles occurs), (2) a busy is placed on the floating point adder for (VL)+4 cycles, (3) a busy is placed on V3 for cycles 1 through 7 and cycles 9 through 7+max((VL),5). Cycle 8 is the "chain" cycle.

The Timing Chart

We can keep track of important cycles by listing them in a timing chart. Then, when we want to consider whether a particular instruction can issue, we have the information at hand. In practice, it is easier to list the cycles when a component will next become ready for use than to record those in which it is busy.

In such a chart, I and C refer to issue cycle and completion cycle for scalars, respectively, while I,C,Ø,F, and R refer to issue cycle, chain cycle, operand register ready cycle, functional unit available cycle, and result register ready cycle for vectors.

```
Thus we have:               I   C   Ø    F    R

      62312   S3   S1+FS2   0   6
```

while, supposing the following instruction comes in sequence with the above and that (VL) = 64:

```
     171312   V3   V1+FV2   1   9   65   69   73.
```

The numbers recorded in the various columns represent the cycles in which certain important changes will occur as a result of the issue of the instruction in question. (Since for scalar instructions, the last three columns are not particularly informative, one may omit them.) Different types of instructions tie up different machine resources for differing numbers of cycles, as indicated in Table 1. (See also Appendices A and D of the CRAY-1 Hardware Manual.) In the examples that follow, we will demonstrate the practical use of these timing numbers. In general, the entry in the C column is the I number plus the appropriate instruction execution-complete time from the first column of Table 1.

Preliminary Considerations

Consider the first add mentioned above: 62312, with I = 0 and C = 6. The 6 has two meanings. First, it is the cycle on which the result will be returned to S3 via the S-register output path. This means that this number cannot appear as the C cycle for any other (later issued) instruction whose result is destined for any S-register. For example, if the next instruction were 76567, transmit a V-register element to S5, which takes 5 cycles, then the machine must delay its issue. If you are recording the I and C numbers for a series of instructions, you should notice when you record two identical numbers in the C column. If the second is a result for the same set of registers as the first, it will be delayed, and you must adjust the issue cycle accordingly.

Secondly, the 6 has another meaning. Cycle 6 is also the cycle on which the register becomes available for use (either as an operand or a result) by another instruction. For example, in coding a set of instructions, one might attempt to reuse an S register before it has completed a previous operation. Thus, one might do a reciprocal into S6 and then read the time clock into S6. The timing is then:

	I	C
70610	0	14
72600	14	15

since the result of the clock read is not allowed to use S6 until the reciprocal is through with it. This assures that the result of the reciprocal will be overwritten by the later instruction.

It is perhaps more common that a later instruction which would use the result of the reciprocal as an operand, would have to wait for it. Thus:

	I	C
70610	0	14
67561	14	21

would be the timing for these two instructions.

For vector instructions, the relations among the numbers I, C, Ø, F, and R, are found as follows: When the issue time I becomes known, then C will be equal to I + the chain time for this instruction (the chain time being the functional unit time + 2), Ø will equal I +(VL), F = I+4+(VL) (thus F will normally be Ø+4) (here, however, one exception exists, for vector store F = I+5+(VL)), and finally R = C + (VL). For short vectors, where (VL) ≤ 4, C and F are as before, while Ø = I+5 and R = C+5.

Thus if (VL) = 2, we have:

	I	C	Ø	F	R
171312	1	9	6	7	14 .

All five vector timing numbers depend only on the chain (C) cycle (from Table 1), (VL), and issue (I).

Two Basic Examples and Comments

In the two examples below, taken from (more or less) real programs, nearly all of the main ideas surrounding accurate timing of code are mentioned. Examine the instruction sequence and refer to the notes for an explanation of the timing numbers listed.

Example 1

First, we consider the earlier example, ABC:

```
*         CFT I=ABCSF,ØN=G,L=LSF,B=BSF
*           LDR I=(BSF,BABCS),ML=MSF,X=XBS,ØRDER=CLNB,FIRST=BABCS
*       XBS
            CØMMØN /ABCØMMØN/ A(56),ØUTRANGE,B(56),C(56)
            CALL LINK('UNIT59=TERMINAL//')
            Y = X*X*X*X*X*X*Y*X
            DØ 1 I = 1,169
      1     A(I) = I
            ØUTRANGE = 600004000000000000000B
            M = IRTC(0)
            CALL ABCS
            N = IRTC(0)
            X = N-M
            WRITE(59,59) C,X
     59     FØRMAT(7F6.0)
            STØP
            END
```

ABC consists of a FØRTRAN part, ABCSF (MAIN.), where the RTC is read, and a CAL part ABCS, where adds are done. We note that we are timing the case where the assembly portion is loaded first.

Listed below is the set of six assembly instructions generated by CFT for the portion of the code where the RTC read occurs (extracted from the long listing). The address listed is after the load. Recall that I and C refer to the machine cycle on which instruction issue and completion, respectively, occur (see Table 1). (The small letters refer to notes following.)

Address	Machine code (octal)		Mnemonics (decimal)		I	C	Comment
251a	072300		S3	RT	0e	1f	Read RTC
251b	130300	000225	M,0	S3	1g	-h	Save RTC
251d	022700		A7	0	3i	4j	Arg count
252a	007000	001000	R	ABCS	4	19k	Call subroutine
252c	120100	000225	S1	M,0	1657m	1668n	Get saved RTC
253a	072700		S7	RT	1659	1660	Read new RTC

Notes: (a,b,c,d at the left refer to the parcel address in the word where the instruction is located.)

e. Assume all resources of the machine are available, initially.

f. A "72" instruction requires one cycle to complete after issue (see Table 1). If any previously issued instruction had needed to put a result into any S-register during cycle 1, the issue of this instruction would have to be delayed by the machine.

g. The instruction following a 16-bit instruction may issue on the next cycle (if there is no conflict, as is the case here), S3 being now available.

h. A store instruction uses an S or A register only during the issue cycle. The result actually reaches memory several cycles later, but for purposes of subsequent fetch instructions, vector loads, or memory busy conditions, the memory is essentially free after four cycles, while the register itself remains free.

i. The instruction following a 32-bit instruction may not issue until after a delay of one cycle (to bypass the lower 16 bits).

j. A "22" instruction requires one cycle to complete after issue. If a previously issued instruction needed to put a result into any A-register during cycle 4, this issue would be delayed. (But an S-reg result could complete then without delaying this.)

k. This instruction, which would normally complete at cycle 18, is delayed for one cycle by memory busy from the previous store, since a memory-busy condition is not allowed when starting the fetch of the next 16-word buffer-load of instructions. If this "007" instruction addressed an instruction from code already in a buffer, it would complete at cycle 9. In the case of a jump instruction, completion means that the jumped-to instruction may issue.

m. This fetch instruction cannot issue until the called subroutine returns to it. See the analysis of ABCS below.

n. When it does issue it will require 11 cycles for the contents of memory to reach the S-register. The memory bank will be free after only four cycles.

Now consider the CAL portion of our example, called by the FORTRAN portion above.

```
                                   *       CAL  I=ABCS,B=BABCS,L=LSC
                                              IDENT             ABCS
                                              COMMON   ABCOMMON
                               71   A         BSS               57
                               70   B         BSS               56
                               70   C         BSS               56
                                              BLOCK             ABCS
                                              ENTRY             ABCS
022100                              ABCS       A1                0
022263                                        A2                51
1211 00000000C                       LOOP     S1                A,A1
1212 00000071C                                S2                B,A1
062312                                        S3                S1+FS2
1313 00000161C                                C,A1              S3
030110                                        A1                A1+1
031012                                        A0                A1-A2
011 00000000c+                                JAN               LOOP
005000                                        J                 BOO
                                              END
```

Since the instructions here form a loop to be performed 51 times, we must consider them more than once. The instructions for pass 1 are:

Address	Machine code (octal)		Mnemonics (decimal)		I	C
200a	022100		A1	0	19k	20
200b	022263		A2	51	20	21
200c	121100	025511	S1	A,A1	21	32
201a	121200	025602	S2	B,A1	23	34
201c	062312		S3	S1+FS2	34p	40q
201d	131300	025662	C,A1	S3	40	-r
202b	030110		A1	A1+1	42	44s
202c	031012		A0	A1-A2	44	46
202d	011000	001002	JAN	LOOP	48t	53u
(203b	005000)		(J	BOO)	(50	57)v

Notes for pass 1:

k. See previous note k.

p. The issue of the add instruction is delayed until both operands (S1 and S2) have arrived from memory. The completion cycle of the S2 fetch is the start cycle of the add.

q. A floating point add requires six cycles to complete (from Table 1).

r. Normally, we don't need to consider memory. S3 is available to start the store at cycle 40, and remains available for other use in the next cycle.

s. An address add requires two cycles. (So does an A to A move, which is really an add of 0.)

t. A conditional jump instruction does not issue until two cycles after the needed operand becomes available. (A0 is returned at 46; 47 is skipped; 48 is issue.) Other instructions, even one using A0 (but not putting a result into A0) could issue at 47, and the jump would still go at 48.

u. This in-stack branch (to 200c) requires five cycles.

v. The numbers here refer to the cycles on which this instruction would have issued and completed, if the program did not branch back.

The instructions and timing for passes 2 and 51 are as follows

Address	Machine code (octal)		Mnemonics (decimal)		I	C
Pass 2						
200c	121100	025511	S1	A,A1	53u	64
	. . .		(add 32 to Pass 1 numbers)			
202d	011000	001002	JAN	LOOP	70	75
(203b			J	B00	72	79)v
	. . .					
Pass 51						
			(add 1600 to Pass 1. number)			
202d	011000	001002	JAN	LOOP	1648	1653
203b	005000		J	B00	1650w	1657x
252c	120100	000225	S1	M,	1657	1668
253a	072700		S7	RT	1659	1660
253b	120200	000225	S2	M,	1660	1672y
253d	120300	000225	S3	M,	1663y	1676

Notes for Passes 2 through 51:

u. The in-stack branch completes and this instruction issues during cycle 53.

v. Once again, these are the "if it didn't" times.

w. This time it doesn't.

x. The return jump requires only seven cycles to complete because the code that called this routine is still in a buffer.

y. Consecutive scalar loads (or stores) may issue as few as 2 cycles apart and, if they do not address the same memory bank, finish in 11 additional cycles. If the second does address the same bank, it will require one or two extra cycles to finish, and a third consecutive scalar load (or store) will be delayed from issue until memory is free (at most four cycles later).

In general, a scalar load or store that encounters a memory conflict (which could come from I/O), issues as usual. This allows subsequent nonmemory instructions to proceed normally, while delaying memory instructions until the conflict is resolved. On the other hand, vector loads or stores (and instruction-buffer loading) wait until memory is entirely free before issuing (or starting). Such delays usually last no more than two cycles.

The cycles listed above are the actual machine cycles on which the events happen for the sequence of instructions given. It should be clear, however, that we could predict these numbers from the timing information in Table 1, together with a minimal understanding of the material from Appendix A (with the exception, perhaps, of the memory conflict details). One simply proceeds line by line, recording the five columns of numbers, left to right.

Thus, given the task of writing an efficient scalar loop to compute C =A+B, we can try a few alternate ways to do it, timing each one as we go, until we have identified the one with the lowest last-issue cycle.

For example, changing the three lines

```
         C,A1    S3
         A1      A1+1
         AO      A1-A2
    to
         A1      A1+1
         AO      A1-A2
         C-1,A1  S3
```

would cut six cycles from the loop time and thus result in nearly a 20% saving in the measured execution time, (26 rather than 32 cycles per loop).

While it is actually possible to accomplish this loop by a scalar method in 14 cycles per pass, the parallel, nonrecursive nature of the loop allows a much greater saving by using vector instructions. So, let us now consider code ABCV, and list its timing details. For an alternate view, we use CIVIC for this compilation.

Example 2

```
                *    CIVIC ABCVF CVF BVF LVF P24 L
                *    LDR I=(BABCV,BVF),ML=MVF,X=XVF
                * XVF
                       CØMMØN /ABCØMMØN/ A(56),ØUTRANGE,B(56),C(56)
000000A                CALL LINK('UNIT59=TERMINAL//')
000001D                DØ 1 I = 1,169
000002C          1     A(I) = I
000010D                ØUTRANGE = 60000400000000000000B
000011D                M = Q8RTC(0)
000012C                CALL ABCV
000013B                N = Q8RTC(0)
000014B                X = N-M
000014A                WRITE(59,59) C,X
                59     FØRMAT(7F6.0)
000037C                STØP
                       END
                            *    CAL  I=ABCV,E=X00,B=BABCV,L=LVC
                                      IDENT        ABCV
                                      CØMMØN       ABCØMMØN
                        71   A        BSS          57
                        70   B        BSS          56
                        70   C        BSS          56
                                      BLØCK        ABCV
                                      ENTRY        ABCV
      022363                 ABCV     A3           51
      0200 0000000C                   A0           A
      002003                          VL           A3
      176100                          V1           ,A0,1
      0200 0000071C                   A0           B
      176200                          V2           ,A0,1
      171312                          V3           V1+FV2
      0200 0000161C                   A0           C
      177030                          ,A0,1        V3
      005000                          J            B00
                                      END
```

Again, we consider the code from one read RTC to the next. Note that since this particular set of adds is not more than 64 in length, it can be done without looping instructions.

We will now record the full five columns of numbers. The I, C, Ø, F, and R refer to issue cycle, chain cycle for vector instructions (or completion cycle for scalars), operand register(s) free cycle, functional unit free cycle, and result register free cycle, respectively.

Address	Machine code (octal)		Mnemonics (decimal)		I	C	Ø	F	R
5013d	072300		S3	RT	0	1			
5014a	130300	005053	M,	S3	1	5			
5014c	022700		A7	0	3	4			
5014d	007000	024000	R	ABCV	4	9e			
5000a	022363		A3	51	9	10			
5000b	020000	000200	A0	A	10	11			
5000d	002003		VL	A3	12	13			
5001a	176100		V1	,A0,1	13f	22g	-h	68i	73j
5001b	020000	000271	A0	B	14k	15			
5001d	176200		V2	,A0,1	68l	77	-	123	128
5002a	171312		V3	V1+FV2	77m	85n	128o	132n	136n
5002b	020000	000361	A0	C	78	79			
5002d	177030		,A0,1	V3	136p	-q	187r	192s	-
5003a	005000		J	B00	137	144			
5015b	072100		S1	RT	144t	145			
5015c	130100	005054	N,	S1	192u	-			

Notes

(a, b, c, and d are parcel addresses, after the load, as before.)

e. For this compilation the destination of the return jump is already loaded into a buffer, so the branch instruction executes in only five cycles.

f. To begin execution, this vector instruction needs A0 and VL to be ready, V1 to be free, and memory to be free. Since they are, it issues.

g. The first result will be arriving from memory nine cycles after the issue cycle. This cycle (cycle 22) is the chain cycle for this memory load. (More on chaining in note m.)

h. When this instruction issues (cycle 13) it transmits as operands the contents of the VL register, the special value 1, and register A0 to the memory functional unit. (Some vector memory loads use a second A-register for the increment.) All these scalar transmissions occur during the issue cycle and are held by the functional unit thereafter. [When A0 and S0 are used as special values their reservation is not checked, and so they do not delay issue. Here, however, A0 is also used to hold an address, and if it had not been free when needed, the issue would be delayed.] For a vector load instruction, no vector register is used as input, so no entry is made in column Ø.

i. For this instruction, the functional unit involved is memory. As with scalar memory references, a memory bank will be busy for four cycles with each word read. If the vector load moves through at least three other banks before returning to a previous one (as is the case here), no conflicts will arise, and a new word will be read each cycle. The first word is requested at cycle 13 and the 51st at cycle 63. The memory will be busy for 4 more cycles, through cycle 67, and free for another memory reference in the next cycle. We record 68 = 13+51+4 under the functional unit free column. Notice that memory is free five cycles before register V1 is ready.

j. When this instruction issues (cycle 13), it puts a hold, or reserve, on register V1 in order to keep it available for the words coming in from memory. The reserve will be lifted after the last word arrives. Since the (VL) is 51, the last (51st) word will arrive in cycle 72. (The first arrives in cycle 22.) In the next cycle the V1 register may be used for another purpose; therefore we record 73 = 22+51 under the result register free column. The CRAY hardware has one element pointer for each V-register, and it is used to select one of the 64 positions in the V-register. The pointer for register V1 is automatically stepped from 1 through 51 during cycles 22 through 72.

k. Since the previous vector instruction read out A0 and (VL), saving them in the functional unit at the start of the vector load, subsequent instructions may modify them immediately without affecting the previous instruction.

l. Here a major delay is encountered. This instruction also transmits words from memory to a V-register. The register is available but the memory is busy, so issue is delayed till it is free (in cycle 68).

m. This instruction chains. At cycle 69, it is first considered for issue. However, before it can begin executing, this vector add needs to have the vector length register, register V1, register V2, the floating point add functional unit, and register V3 free. V1, as noted, becomes free at cycle 73; V2 will not be free until 128; but the first element will arrive at cycle 77 and during that one cycle, it can be redirected, or chained, to serve as input to the add unit as well as being put into V3. The conditions for chaining are thus satisfied during cycle 77, and so the instruction issues.

n. The first result exits from the floating point adder eight cycles after the first operands were sent over. For this instruction, then, its chain cycle is 85 = 77+8. Similarly its result register (V3) free cycle is 136 = 85+51, and its functional unit free cycle is 132 = 77+51+4. The four extra cycles here are equivalent to the four extra cycles needed for memory free by the memory functional unit. All functional units remain reserved for four extra cycles after the last element arrives during vector instructions. This means that a subsequent scalar (or vector) floating point add cannot issue until cycle 132, since it shares this unit.

o. Since this instruction requires that vector register operands be sent to the adder for the next 51 cycles, a reserve is placed on registers V1 and V2 until cycle 128, at which time they will both be free and able to be used by a subsequent operation.

p. This vector store does not chain from the add. In the first place, at cycle 85, the chain cycle for V3, the memory is busy completing the load of V2. In the second place, store instructions are barred by the hardware from chaining even if the memory functional unit is free. The store doesn't begin at cycle 123 (when the memory becomes free) either. It can't issue at 123 because the element pointer for V3 is not pointing to V3's first element, which the store needs, but rather at element 39, which is being returned by the floating-point adder. It finally issues when register V3 is not otherwise busy and can have its element pointer reset, namely cycle 136, the result register free cycle for the earlier add.

q. A store doesn't chain to anything, either.

r. Register V3 will be free after the store at cycle 187 = 136+51.

s. Finally, the memory functional unit will become free from the store five cycles after the operand register, V3, is free. All other instructions free their functional units four cycles after their operand registers but store requires one extra cycle.

t. Since the return from subroutine did not require memory, as the address is already in a buffer, the next instruction, which for CIVIC is the read of the RTC, gets issued well before the vector store completes.

u. Finally, we note that the final store of the RTC value to memory is delayed by the memory busy condition from the vector store, and issues when the memory functional unit ready cycle occurs.

Conclusions

It should be clear from the timing chart above that the CRAY-1 is not really very busy during this vector add routine. For example at cycle 78, its busiest cycle, V-registers 0,4,5,6, and 7 are free along with the shift, fixed add, multiply, reciprocal, and logical functional units. Moreover, the next 55 cycles (as well as most of the previous 60) could be used to issue independent instructions for a related calculation, if one needed to be done. (In fact, we can actually decrease the time for ABCV by four cycles by using some of the idle resources.)

Frequently, parallel use of available resources can be made, especially in the case of vector loops. Three examples of actual code are presented in Section V to show this: ZVSPEK, QVDIVC, and QVSQRTH.

IV. THE COMPUTER CODE CYCLES

CYCLES is a public file on the CRAY-1 computers at LLNL. It was written by Rollin Harding. A Fortran version of it has been made available to Cray Research Incorporated and is being modified for use under their system.

CYCLES is not a simulator and does not have knowledge of the values in all machine registers. It does, however, try to keep track of the values in the VL and A registers. Options allow these register values to be specified for CYCLES' use.

The rest of this section is taken from the documentation for CYCLES. A full writeup, CYCLEWUP, can be extracted from the CYCLES public file using LIB.

Cycles Writeup

CYCLES was designed for detailed analysis of instruction scheduling in compiled or assembled CRAY codes. The timing analysis is presented in the spirit of Harry Nelson's report, UCID-30179, Rev. 1, "Timing Codes on the CRAY-1". Harry supplied additional timing details and tested the code extensively during the debugging period.

Input to CYCLES is any HSP file from CAL, CIVIC, CFT, or DDT which contains the machine code listing. CYCLES accepts single or double column listings from CIVIC (M or L option) and the four instructions per line format from CFT (on=g). Sequences of octal parcels may be entered from TTY or by specifying octal word limits in a controllee or other binary file. In TTY or binary modes CYCLES adds the equivalent CRAY assembly language instructions to the output, i.e. does a CRAY UNDO. CYCLES will also accept the history file produced by DDT in the MNE output format mode. This form has the advantage of using correct symbols for variables in the program being undone.

Output consists of a copy of the input file with up to seven columns of timing information added for each machine instruction line. (This overwrites the comment field in CAL listings.) The NOCOPY. option will suppress most non-instruction lines from being output. The seven timing columns are:

W	number of cycles this instruction waited to issue
D	octal codes identifying any delays
I	issue cycle for the current instruction
C	vector chain cycle or scalar completion cycle
O	vector operand register ready time
F	vector functional unit ready time
R	vector result ready time

The I column is always given; others are suppressed if null or irrelevant for the current instruction. Alternate definitions for columns C, O, F, and R for jump instructions are given below.

CYCLES is very fast and is easily run as a controllee under TRIXGL. Output can be viewed without line wraparound by using TUBE command S or TRIXGL command TV,1 for small characters. Effects of altering instruction sequences can be checked easily by rearranging lines in CYCLES' infile and rerunning it without reassembling your code. One may also rearrange lines in CYCLES' outfile and then use that as the infile. CYCLES CIVIC output is compatible with single column CIVIC output. CYCLES' CFT, CAL, and binary output are compatible with CAL output. CYCLES' DDT output is compatible with DDT output.

Abilities and limitations

CYCLES is aware of most of the fine points of CRAY instruction scheduling:

- chaining requirements
- recursive vector operations
- no waits for special A0 and S0 operands
- memory functional unit requirements
- vector memory conflicts due to 8*n increments
- A and S register trunk conflicts.
- extra delay after A0 or S0 ready for conditional jumps.
- scalar memory bank conflicts (with limitations)
- instruction buffer fetches, conflicts, and delays.
- other special cases

CYCLES has to make assumptions about loader dependent conditions such as instruction buffer delays and scalar memory bank conflicts. Bank conflicts may not be detected if memory addresses are indefinite. Addresses are indefinite if they involve undefined A register values or unspecified relocation flags. Options are provided to specify that the current code block (local relocation) is loaded on a 20b-word buffer boundary or that all external blocks (subroutines or commons) are loaded on 20b-word boundaries. The relevant option names are +., x., and +x. to set relocation flags, and MBOFF. to turn off bank conflict checking. IBOFF. turns off instruction buffer checking.

VL and A registers

Many instruction timings depend on values of the vector length register and A registers. CYCLES attempts to keep VL and A regs current as instructions are processed that set those registers. A registers set from memory or from S registers are considered indefinite. Results of A register calculations involving indefinites are also indefinite. VL will be set to 64 if it is set from an indefinite A register. Register changes are reported in the output. Automatic register setting can be disabled by the NØVLA. execute line option.

You may explicitly reset values for VL, A, or NI (next issue) by inserting control lines into CYCLES' input file or as comments in a CAL source file. In column 1 of the input file use Ln to set VL to n (decimal), use Cn to reset counters and force the next issue to cycle n (decimal), and use An,m to set register An to m (decimal). CAL comments *Ln, *Cn, and *An,m would have the same effects.

Jump instructions

For conditional jumps, CYCLES assumes drop through timing. Normally, the cycle counter is reset to zero after each unconditional jump. However, if the following instruction is recognized (by its address) as the target instruction, then timing continues without reset. This can be accomplished by control cards (CYCLE ØFF/IN/ØUT or REPEATn described below) or by rearranging the input file.

For a jump instruction certain columns are redefined:

C	Earliest issue for the jump target if the jump is taken
O	Target instruction buffer code (see I-buff section)
F	Target issue time for an in-buffer jump
R	Target issue time for an out-of-buffer jump

An out-of-buffer jump can be significantly delayed if memory is busy, for instance, completing a vector store.

You can control the output for a jump to a later instruction by inserting a control line CYCLE ØFF immediately after the jump and a CYCLE IN or CYCLE ØUT line immediately before the target instruction. CYCLES will stop timing after the ØFF and will resume by issuing the target instruction at the proper IN buffer or ØUT of buffer issue time. Comments, *CYCLE ØFF, etc., can be used in a CAL source as well.

A REPEATn line can be used for continuous timing over a jump to an earlier instruction. The REPEAT line is inserted immediately before the target instruction. From then on, each jump instruction is checked to see if its target has an active repeat line. If it does, the count n is decremented, and timing continues at the target line using the in buffer time plus any appropriate delays for registers or functional units. Up to ten repeat lines may be active at any time. Repeats may be nested.

Instruction buffer (I-buff) delays

The CRAY has 4 instruction buffers. They are loaded in rotation. Each holds 20b words (64 parcels) of instructions. I-buff delays occur each time execution shifts from one buffer to another due to a jump instruction or simply when crossing from one 20b block to the next. Additional delays result when memory operations conflict with instruction fetches or when a two-parcel instruction straddles a buffer boundary. For I-buff checking CYCLES assumes that relative word zero is loaded on a 20b-word boundary.

I-buff delays are indicated in the usual way, using delay code 200b, but additional information is also given:

- The first instruction from a buffer is marked (between the W and D columns) by a letter a,b,c, or d for buffer 0,1,2, or 3. Upper case means the instructions were fetched from memory; lower case means the buffer was already loaded.

- For jump instructions the target instruction buffer is given under the θ column. Again, upper case is out-of-buffer; lower case is in-buffer. A jump to an external (x reloc) address is always considered out-of-buffer. An unconditional jump out-of-buffer clears one instruction buffer unless the NOICLR. option is used. A Bn line can be used to clear n additional instruction buffers.

- Delay code 10000b shows that an instruction fetch was delayed because memory was busy. Because of look-ahead, this does not cause an immediate delay of issue, but it does signal a possible delay for a subsequent issue (usually the target instruction of an out-of-buffer jump appearing in column F).

- Delay code 20000b indicates that the parcel address for the current instruction was not in a current I-buff or one that had been fetched. No delay is assessed.

- Delay code 40000b indicates the possibility of a delay that this version of CYCLES couldn't determine. The marked instruction is parcel 17c of the current instruction buffer. If the next instruction (17d) happens to be a two-parcel instruction (this is what the timing subroutine didn't know) then 17c would be delayed until one cycle before the issue time indicated on the next line for 17d. This delay of parcel 17c could cause further delays not shown for 17d, 20b, or later instructions. Correct timing can be produced in the current version by inserting an "In" control card before 17c, where n (decimal) is the correct issue time for 17c.

Availability of CYCLES

The latest version of CYCLES is maintained in CRAY public file CYCLES. The HELP packages are reproduced below. The output file is named Hinfile and is left on disk. An existing file will be overwritten. If the file overflows, sequence numbers will be added: 00, etc.

This writeup is available as CYCLEWUP in public file CYCLES. It will be revised as suggestions are made or changes made to CYCLES. The revision date is given on line 1.

Please send suggestions for enhancements to CYCLES or listings of any bugs you encounter to Rollin Harding in A-Division (L-16).

```
CYCLES HELP:

      execute lines:
    cycles hspfile type <nocopy. novla.  ...  noiclr. &> / t v
    cycles tty / t v
    cycles binfile fwa lwa <abs. end> / t v  (binary input mode)
      type is  cal  cft  civic  or  ddt
      <> shows options. keep in order. no comma for dropouts.
      nocopy.  suppresses non-instruction lines
      novla.   defeats automatic setting of vl and a registers
      mboff.   suppresses mem bank conflict checking
      +x.      assumes both +. and x. (increases mem bank checking)
      +.       assumes present routine is loaded on a 20b boundary
      x.       assumes externals are loaded on 20b-word boundaries
      +xreloc. oct  sets both +reloc. and xreloc. (affects i-buff chks)
      +reloc. oct  =>offset=oct for local word 0 in i-buff and mem bank
      xreloc. oct  =>offset=oct for external reloc vars and subrs
      ibuff.   suppresses instruction buffer checking
      noiclr.  suppress clearing an i-buff after out-buf uncond jmp
      &        to continue execute line
      fwa,lwa  are octal; may have a,b,pa,pb,etc. parcel tags
      abs.     changes assumed 3400b minus word offset to 0
      end      says don't ask for additional  fwa lwa  pairs
      outfile name will be  h+infile name
    type  delayed  for list of delay codes
    type  helpcc   for list of infile control card options


HELPCC:

      in col 1 of cycles' input file (cal,civic,cft,ddt) use:
    ln          to set vector length to n (decimal)
    cn          to reset registers and set next issue time to n (decimal)
    bn          to clear n additional instruction buffers
    in          to set next issue to n (dec) without resetting registers
    am,n        to set register am to value n (decimal)
    repeat n    before target instr to time n jumps back to target
    cycle off   disable cycle counting, use after conditional jump
    cycle on    resume counting at the 'in buffer' jump time
    cycle in    same as cycle on
    cycle out   resume counting at the 'out of buffer' jump time
      use any of these as comments in your cal infile: *am,n etc.
      in TTY mode use  ln cn in an,m  as above, and use
    ploci       to set parcel to word 'loc' and parcel i=a,b,c,d,pa,
```

Table of Delay Codes

DELAYCD:

octal delay codes:

1b	functional unit not ready
2b	result register not ready
4b	operand register not ready
10b	waiting for chain cycle
20b	a or s register trunk conflict
40b	scalar memory operation bank conflict
100b	conditional jump delayed by a0 or s0 busy last 2 cycles
200b	instruction buffer delay
400b	operand chain cycles don't match. can't chain.
1000b	missed chain cycle
2000b	waits for all instructions to complete
4000b	waiting for register block transfer to finish
10000b	instruction fetch delayed by memory busy
20000b	current instr in unexpected buffer. no delay added
40000b	possible two parcel split delay of 17c

V. EXAMPLES

As mentioned above, source code for all these examples is in public LIB file CLASS on the CRAY-1. The timing numbers are from code CYCLES.

ZVSEEK

ZVSEEK is a BASELIB function designed to find a target value in an unordered list. The original version was written about a year before the LLNL machine arrived and has since been upgraded by use of timing analysis to run more than twice as fast. Most of the speed increase was obtained through a simple algorithm change: replacement of a logical vector instruction by a fixed add. However, an additional healthy gain came through improved handling of the vector looping technique. The main loop of the original routine consists of 10 instructions.

Main Loop of ZVSEEK (Øld Version).

This version prestores the target at the end of the search array, so that it must eventually exit on a hit.

Timing of original version: VL = 64.

Address	Instruction		I	C	Ø	F	R	Comment
L64	V0	,A0,1	0	9	-	68	73	Get next 64 values
	V1	S4\V0	9	13	73	77	77	XØR each with target
	VM	V1,Z	77a	-b	141	145	147c	Check for hit
	S1	VM	147c	148				VM to S for count
	S0	VM	148	149				VM to S for test
	A4	ZS1	149	152				Count left zeroes (needed if hit)
	JSN	HIT	151d	156				Exit if hit
	A0	A5+A6	153	155				.LØC. of next 64 values
	A5	A5+A6	154	156				Up A5 by 64
	J	L64	155	160e				Go check next 64 values

Notes:

a. Since the VM is set by the logical functional unit, this instruction, which also uses the logical unit, delays until the unit is free and does not chain.

b. The vector mask instruction never chains its output to anything.

c. While another logical vector operation using the VM-register could start at cycle 145 (for example, merge), the VM cannot be read out to an S-register until two cycles later (see the CRAY-1 Hardware Manual, p. 4-69 or page 122 of the online version, LCSD-158). Thus, we record 147 as the register free cycle.

d. This instruction is delayed one cycle since S0 has not been ready for the necessary unused cycle.

e. As written, this loop is taking 160 cycles for each 64 elements searched.

<u>Improved Version with XOR Replaced by Fixed Subtract</u>

Address	Instruction		I	C	O	F	R	Comments
L64	V0	,A0,1	0	9	-	68	73	
	A0	A5+A6	1	3				No reason to wait
	V1	S4-V0	9	14	73	77	78	Subtract each from target
	A5	A5+A6	10	12				Get it out of the way
	VM	V1,Z	14f	-	78	82	84	
	S0	VM	84g	85				
	S1	VM	85	86				
	A4	ZS1	86	89				
	JSN	HIT	87	92				
	J	L64	89	94h				

Notes:

f. Since the fixed subtract was used in place of the logical difference, the vector mask instruction can now chain its input operands.

g. Exchanging the order of the VM transmits to S saves a cycle later on.

h. The loop is now performing the same service as before but using only 94 cycles for each 64 elements searched.

This latter loop represents approximately a 40% improvement over the former. However, because: (1) no functional unit is used for more than 68 cycles, (2) no register is used for more than 73 cycles, and (3) there are plenty of unused registers, one would expect that additional savings may be possible.

Another item that should be taken into consideration is that this method is rather inefficient for those searches in which the target value is found in the first portion of a set of 64 elements searched. For example, suppose the list we are searching has 64 entries. On the average, we would expect to find the target value in the first half of the list as often as in the last half, but for all these cases, the loop as written will require the full list to be tested.

In fact, there is a clever (almost heroic) method available which can go through this particular search loop in exactly 68 cycles per 64 elements searched. The treatment below, however, is somewhat easier to code (and debug) and offers an improvement in the time used to find the target over even the heroic method, on the average, for searches up to 512 in length.

The main tricks employed are: (1) breaking the array into vectors of length 32 each; (2) replicating the loop but using a different set of V-registers for each half, (3) loading and subtracting a second set of 32 elements while waiting for the VM instruction for the first 32 to finish, and (4) loading extra unneeded elements in the first half of the loop and using an otherwise unneeded vector operation in the second half to maintain the correct timing so that the load-subtract-VM chain will not be broken.

The timing chart for the main loop is given below. The notes following are referenced by line number.

	Address	Instruction		I	C	Ø	F	R
1	First half of main loop							
2								
3		A2	35					
4		A5	ADDRESS					
5		S4	TARGET,					
6								
7	L64	A0	A5	0	2			
8		VL	A2	1	2			
9		V0	,A0,1	2	11	-	41	46
10		A6	32	3	4			
11		S6	A6	4	6			
12		VL	A6	5	6			
13		A6	A4	6	8			
14		V1	S4-V0	11	16	43	47	48
15		S1	VM	15	16			
16		VM	V1,Z	16	-	48	52	54
17		S0	S1	17	18			
18		A4	ZS1	18	21			
19		JSN	HIT	20	25			
20		S0	S6-S3	22	25			
21		A6	32	23	24			
22		A5	A5+A6	24	26			
23		S3	S3+S6	25	28			
24		JSP	DUN	27	32			
25								

```
26     Second half of main loop
27
28                     A0    A5          29         32
29                     V2    ,A0,1       41         50          -        77        82
30                     V3    S4-V2       50         55          82       86        87
31                     A4    15          51         52
32                     S1    VM          54         55
33                     VM    V3,Z        55         -           87       91        93
34                     S0    S1          56         57
35                     VL    A4          57         58
36                     A4    ZS1         58         61
37                     V0    V6<A0       59         65          74       78        80
38                     JSN   HIT         60         65
39                     S0    S6-S3       62         65
40                     A5    A5+A6       63         65
41                     S3    S3-S6       64         67
42                     A4    32          65         66
43                     JSM   L64         67         72
44
45     DUN             A5    A5-A6       69         71
------------------------------------------------------------------------------
```

Notes:

Line 8. Although we are only going to check 32 elements, we take care to load 35. The reason for this will appear at line 33.

line 9. Since there are 35 elements being loaded, F = 2+35+4.

Line 12. Now we cut the VL back to 32. Reducing the vector length in the middle of a chain is perfectly safe. However, increasing it while chaining can lead to wrong answers (i.e., the answers may differ depending on external happenings such as I/O activity, system interrupts, and operands out of range).

Line 16. The chain continues, with the functional unit becoming free at cycle 52, while the VM itself is not transmittable to S1 until 54.

Line 29. When we reach here we are simply waiting for the previous vector mask instruction at line 16 to finish. Since the memory functional unit is free, we may as well start to load the next 32 elements. We choose not to load 35 elements this time.

Line 30. The fixed adder is also free so we may as well start the next subtract at chain time.

Line 32. We must rescue the previous VM register setting before we can form a new one. Cycle 54 is the earliest this can be done.

Line 33. The cycle following the move of the VM to S1 is the first cycle in which we can start a new VM instruction. Happily, cycle 55 is also

the chain cycle for the subtract at line 30, so the chaining continues. Notice what would have happened if we had loaded only 32 elements at line 9. First, for that instruction, the functional unit would have gone free at cycle 38. Second, the load at line 29 would have then begun at cycle 38. Third, the subtract at line 30 would have chained at cycle 47. Finally, the VM at line 33 would have missed the chain cycle (52), since we had to hold it up for the move of the old VM to S1. Thus, it would not have issued till cycle 87.
But since the loop will normally continue back to the VM at line 16, and since we have not loaded 35 elements this time, we must do something to hold back the load at line 9 in the next pass, or the VM at line 16 will again miss its chain cycle.

Line 35. Here we start to pull another trick, which will delay the load at line 9 in the next pass and at the same time protect this loop against a problem (in timing, not correctness) that may arise if there is an interrupt during its execution. The protection is free in terms of the cycles required to do it, but it does require extra instructions.

Line 37. This is the protection instruction. Since it is putting 15 results into V0 using the shift functional unit, which has a chain time of 6, it will tie up register V0 until cycle 80. This in turn will cause the next load at line 9, which uses V0, to be held until cycle 80. This is the exact cycle desired, since it will bring the chain cycle from the subtract at line 14 to cycle 94, the cycle immediately after the one in which we can first save the VM (93). At the same time, regardless of whether or not some interrupt has come along and bollixed our careful timing, this will force the next load (at line 9) to hold long enough relative to the previous VM so that we will be back in synch thereafter.

Line 38. In this program address HIT has already been put into an instruction buffer. If this were not the case, the jump would complete at cycle 91.

Lines 37 through 40.
Several instructions are completing in cycle 65; each uses a different register set.

Line 43. After jumping back, we will be holding at line 9 for the completion of the instruction at line 37. The loop time will be 78 cycles for each 64 elements tested, but, on the average, we will exit in the upper half of the loop half the time, which provides a further speed increase, especially valuable for short arrays.

QVDIVO

As another example, we present the coding for QVDIVO, the CRAY-1 STACKLIB divide routine.

On the CRAY, the vector divide algorithm used to accomplish the FORTRAN vector statement C = A/B, where A, B, and C are vectors with arbitrary (linear) stride, requires three vector memory operations, three vector multiply operations, and one vector reciprocal approximation instruction for each 64 elements. The current CFT implementation of the general vector divide loop requires 445 cycles per 64 elements stored plus some startup time, which brings the cost for such a divide to roughly 7 cycles per element. However, by overlaying the storing of the result for the first pass through the loop and the loading of the operands for the third pass through the loop with the multiplying still being carried out for the second pass, one can expect to achieve something on the order of twice CFT's performance. In fact, the theoretical minimum, 205 cycles (68 + 68 for loads + 69 for store) per 64 elements (after suitable startup time) is achieved in this routine. The timing chart for the main loop is given with notes below.

Line	Instruction		I	C	Ø	F	R
-3	V6	V2*IV1	-137	-128	-73	-69	-64
-2	V4	V1*FV6	-64	-55	0	4	9
-1	A0	S5	-63	-62			
0	JSP	TWØTRIP	-62	-48			
*				B U F F E R	B Ø U N D A R Y		
1	V2	,A0,A5	-48	-39	16	20	25
2 LP	VL	A4	-44	-43			
3	A3	A5*A7	-43	-37			
4	S3	A2	-42	-40			
5	V1	S0+V5	0	5	64	68	69
6	S3	S3<6	1	3			
7	S2	S3+S2	3	6			
8	V6	V7*IV1	5	14	69	73	78
9	S3	A3	6	8			
10	S5	S5+S3	8	11			
11	A0	S5	11	12			
12	V0	,A0,A5	20	29	-	88	93
13	VL	A7	21	22			
14	V3	V4*RV2	73	82	137	141	146
15	VL	A4	74	75			
16	A0	S2	75	76			
17	V7	,A0,A2	88	97	-	156	161
18	V5	/HV7	97	113	161	165	177
19	V2	V0&V0	137	141	201	205	205
20	V4	V1*FV6	141	150	205	209	214
21	VL	A7	142	143			
22	A0	S6	143	144			

23	A3	A6*A7	144	150			
24	,A0,A6	V3	156	-	220	225	-
25	S3	A3	157	159			
26	S0	S1-S4	158	161			
27	S1	S1-S4	159	162			
28	A7	A4	160	162			
29	S7	S4	162	163			
30	S6	S6+S3	163	169			
31	JSN	LP	164	169			

Notes:

Line -3. We choose to begin the timing chart somewhat before the loop. We have to start the timing somewhere. Arbitrarily, we may take the start of this instruction as any cycle. Cycle -137 will be convenient.

Line -2. At this point, it is clear that the state of the machine prior to line -3 will have no effect on the issue time of this instruction. (Actually, a vector reciprocal instruction whose result register was V4 could still be in progress and would delay this issue by a few cycles, but that is not the case.)

Lines 0 and 1.
The jump here to TWOTRIP is not taken. However, a 16-word buffer boundary (20 octal) occurs after the JSP instruction, and this delays the next instruction until the new buffer can be loaded from memory. Notice that the time of issue of the instruction at line 1 after the buffer load is the same as it would have been had a jump been taken to it.

Line 5. This move instruction is the first vector instruction in the loop. We have arranged to make it issue at cycle 0. It will wait to issue until V1 has delivered all the operands for the multiply instruction at line -2.

Line 8. This multiply chains with the fixed add (move) at line 5. We have insured chaining by delaying the move long enough to have the multiply functional unit free from line -2.

Line 12. This load will issue as soon as the previous one at line 1 releases the memory (cycle 20).

Line 14. V2, V3, and V4 have been available for many cycles before this instruction can issue. It has to wait for the use of the multiply unit. Note also that the A-register multiplies do not interfere with the floating-point multiplies since they are done in a separate functional unit.

Lines 17 and 18.

These instructions chain.

Line 19. This is another move instruction. The release of V0 by this instruction determines the length of the loop (205 cycles).

Line 20. This does not chain with the move at Line 19. It issues at cycle 141 because it can't get at the multiply unit from line 14 before then.

Line 24. This is the final store instruction. It is released for issue by the availability of the memory from line 17. The memory functional unit also determines the time for the loop since we are using it for 68+68+69=205 cycles.

The following is a timing and accuracy test for QVDIVO:

```
*        RCFT I=TESTQVD,ON=G,B=BQVD,C=COO
* LDR I=(DQVDIV,BQVD),X=XQVD,ORDER=CLNB,FIRST=BQVDIV
*   XQVD
        COMMON /QVCOM/ X(48000),W(48000),U(48000),Z(48000)
        CALL LINK('UNIT59=(TTY,TEST)//')
        DO 3 L = 1,12000,64
        DO 2 I=1,3*L+1
        Z(I) = 4+L
    2   U(I) = 4+1
        K = IRTC(0)
        CALL QVDIVO(W(1),U(1),Z(2),L,4,3,2)
        N = IRTC(0)
        N = N-K
        K = IRTC(0)
        DO 1 I=0,L-1
        X(4*I+1) = U(3*I+1)/Z(2*I+2)
    1   CONTINUE
        M = IRTC(0)
        M = M-K
        DO 4 I=0,L-1
        IF(X*I+1).NE.W(4*I+1)) GO TO 5
    4   CONTINUE
        WRITE(59,60) L,M,N
     60 FORMAT(16,2I6)
    3   CONTINUE
        STOP 1
     5  CONTINUE
        WRITE(59,59) W(4*I+1),X(4*I+1)
        WRITE(59,61) (W(I),I=1,4*L-3,4),(X(I),I=1,4*L-3,4)
    59  FORMAT(3E18.14)
    61  FORMAT(3022)
        STOP
        END
```

QVSQRTH

We conclude our examples with the code for QVSQRTH, a half precise (28-bit-accurate) square root routine for arrays, available in STACKLIB. The full-precision routine QVSQRT is quite similar, requiring one additional iteration but needing, also, a full precision divide during this final iteration. The code is perhaps remarkable in that maximum speed is obtained by breaking the array up into vectors of length 31, and because every vector operation is chained to the previous one. A total of 21 consecutive chained vector operations occur.

Essentially, the idea is to compute an initial guess X0 and then to iterate three times by the formula: Xi+1 = (Xi + Y/Xi)/2, where Y is the number whose square root is desired. The iterative loop can be managed by the four CAL instructions:

```
V0  /HV1
V2  V0*FV3
V4  V2+FV1
V5  S4+V4
```

The halving operation is performed by adding minus one to the exponent. Chaining will end for long vectors at the (+F) instruction since there will be a conflict over the use of register V1. However, by adding one auxiliary NO-OP instruction (a shift of zero), we can achieve the following timing for vectors of length 31, since the +F is delayed until V1 is free.

	I	C	O	F	R
V0 /HV1	0	16	31	35	47
V6 V0*FV3	16	25	47	51	56
V2 V6>A7	25	31	56	60	62
V4 V2+FV1	31	39	62	66	70
V5 S4+V4	39	43	70	74	75

Now, at cycle 43, we can issue another reciprocal operation (to register V7) and continue the procedure without any breaks in the chain. Moreover, since the initial guess can be generated by a similar set of chained operations, the entire calculation may proceed from the initial load, with each successive vector instruction issuing at the chain cycle of the previous one. (In the full-precision routine, the chain is broken during the calculation of the full-precision reciprocal.)

The timing chart for this half-precise square root is given next (for the main loop). A full iteration begins at label ITER. The complete routine is available in file CLASS.

Address	Instruction		I	C	O	F	R
LOOP	V5	V0*FV1	0	9	31	35	40
	V6	V5>A7	9	15	40	44	46
	A7	24	10	11			
	A7	V6+FV7	15	23	46	60	54
	A0	A1+A7	17	19			
	A7	A7-A6	18	20			
	V3	S5+V2	23	28	54	58	59
	S0	+A7	24	26			
	A7	-A7	25	27			
	S7	VM	26	27			
	V4	/HV3	28	44	59	63	75
	JSP	NOLOD	29	34			
	VL	A7	31	32			
	V0	,A0,A3	32	41	39	43	48
	VM	V0,Z	41	52	48	52	54
	VL	A6	42	43			
NOLOD	V5	V4*FV1	44	53	75	79	84
	A7	0	45	46			
	V0	V5>A7	53	59	84	88	90
	S7	S6&S7	54	55			
	A7	A6*A3	55	61			
	S2	VM	56	57			
	V6	V0+FV3	59	67	90	94	98
	S2	S2>24	60	62			
	S7	S2!S7	62	63			
	VM	S7	63	66			
	A1	A1+A7	64	66			
	V2	S4!V6&VM	67	71	98	102	102
	A0	A6-A5	68	70			
	A7	A6*A4	69	75			
	A5	A5-A6	70	72			
	V7	S5+V2	71	76	102	106	107
	JAP	DUN	72	77			
	A0	A6-A5	74	76			
	JAP	SHORT2	78	83			
ITER	A1	A1	80	82			
	A0	A1	82	84			
	V1	,A0,A3	84	93	115	119	124
RTN2	V0	S1*FV1	93	102	124	128	133
	S2	>2	94	95			
	S2	S2>15	95	97			
	V2	S2!V0	102	106	133	137	137
	V3	V2>A0	106	112	137	141	143
	A0	A2	107	109			
	A2	A2+A7	108	110			
	V4	S3+V3	112	117	143	147	148
	V5	/HV4	117	133	148	152	164
	,A0,A4	V7	118	-	149	154	-

```
            V6      V5*FV1      133      142      164      168      173
            A7      24          134      135
            VL      A7          135      136
            VM      V0,Z        137      165      161      165      167
            VL      A6          138      139
            A7      0           140      141
            V2      V6>A7       142      148      173      177      179
            V3      V2+FV4      148      156      179      183      187
            V7      S5+V3       156      161      187      191      192
            V0      /HV7        161      177      192      196      208
            J       LOOP        162      167
------------------------------------------------------------------------------
```

APPENDIX A. AN ABRIDGEMENT OF THE SUMMARY OF CPU TIMING INFORMATION FURNISHED BY CRAY RESEARCH INC.

When issue conditions are satisfied, an instruction completes in a fixed amount of time. Instruction issue may cause reservations to be placed on a functional unit or registers. Knowledge of the issue conditions, instruction execution times and reservations permit accurate timing of code sequences. Memory bank conflicts due to I/O activity are the only element of unpredictability.

SCALAR INSTRUCTIONS

Four conditions must be satisfied for issue of a scalar instruction:

1. The functional unit must be free. No conflicts can arise with other scalar instructions. However, vector floating point instructions reserve the floating point units. Memory references may be delayed due to conflicts.

2. The result register must be free.

3. The operand register must be free.

4. Issue is delayed 1 clock period if a result register group input path conflict would exist with a previously issued instruction. One input path exists for each of the four register groups (A, B, S and T).

Scalar instructions place reservations only on result registers. A result register is reserved for the execution time of the instruction. No reservations are placed on the functional unit or operand registers.

A transmit scalar mask instruction to Si (073) instruction is delayed by (VL) + 6 clock periods from the issue of a previous vector mask (175) instruction, and is delayed by 6 clock periods from the issue of a preceding transmit (Sj) to VM (003) instruction.

Execution times in clock periods are given below. An asterisk indicates that issue may be delayed because of a functional unit reservation by a vector instruction. Memory may be considered a functional unit for timing considerations.

(A=A-register, M=Memory, B=B-register, S=S-register, I=Immediate, C=Channel, T=T-register, V=V-register, * see previous page)

24-bit results:

A<--M	11*	A<--C	4
M<--A	1*	A<--A+A	2
A<--B	1	A<--AxA	6
B<--A	1	A<--pop(S)	4
A<--S	1	A<--lzc(S)	3
A<--I	1	VL<--A	1

64-bit results:

S<--M	11*	S<--S+S	3
M<--S	1*	S<-S(f.add)S	6*
S<--T	1	S<--S(f.mult)S	7*
T<--S	1	S<--(r.a.)S	14*
S<--I	1	S<--V	5
S<-S(log)S	1	V<--S	1
S<-S(shift)I	2	S<--VM	1
S<-S(shift)	3	S<--RTC	1
S<--S(mask)	1	S<--A	2
RTC<--S	1	VM<--S	3

Vector Instructions

Four conditions must be satisfied for issue of a vector instruction:

1. The functional unit must be free. (Conflicts may occur with vector operations.)
2. The result register must be free. (Conflicts may occur with vector operations.)
3. The operand registers must be free or at chain slot time.
4. Memory must be quiet if the instruction references memory.

Vector instructions place reservations on functional units and registers for the duration of execution.

1. Functional units are reserved for (VL)+4 clock periods. Memory is reserved for (VL)+5 clock periods on a write operation, (VL)+4 clock periods on a read operation.

2. The result register is reserved for the functional unit time +(VL+2) clock periods. The result register is reserved for the functional unit +7 clock periods if the vector length is less than 5. At functional unit time +2 (chain slot time) a subsequent instruction, which has met all other issue conditions, may issue. This process is called "chaining." Several instructions using different functional units may be chained in this manner to attain a significant enhancement of processing speed.

3. Vector operand registers are reserved for (VL) clock periods. Vector operand registers are reserved for 5 clock periods if the vector length is less than 5. The vector register used in a block store to memory (177 instruction) is reserved for (VL) clock periods. Scalar operand registers are not reserved.

Vector instructions produce one result per clock period. The functional unit times are given below. The vector read and write instructions (176, 177) produce results more slowly if bank conflicts arise due to the increment value (Ak) being a multiple of 8. Chaining cannot occur for the vector read operation in this case.

If (Ak) is an odd multiple of 8(*), results are produced every 2 clock periods. If (Ak) is an even multiple of 8(*), results are produced every 4 clock periods.

Memory must be quiet before issue of the B and T register block copy instructions (034-037). Subsequent instructions may not issue for 14+(Ai) clock periods if (Ai).NE.0 and 5 clock periods if (Ai)=0 when reading data to the B and T registers (034,036). They may not issue for 6+(Ai) clock periods when storing data (035,037).

The B and T register block read (034,036) instructions require that there be no register reservation on the A and S registers, respectively, before issue.

Branch instructions cannot issue until the A0 or S0 operand register has been free for two clock periods. Fall-through in buffer requires two clock periods. Branch-in-buffer requires five clock periods. When an "out of buffer" condition occurs the execution time for a branch instruction is 14 clock periods. (18 clock periods for 8-bank phasing option.)

A two parcel instruction takes two clock periods to issue.

Instruction issue is delayed 2 clock periods when the next instruction parcel is in a different instruction parcel buffer. Instruction issue is delayed 12 clock periods if the next instruction parcel is not in an instruction parcel buffer.

----

* Multiple of 4 for 8 bank phasing option.

HOLD MEMORY

A delay of 1, 2, or 3 CP will be added to a scalar memory read if a bank conflict occurs with rank C, B, or A, respectively, of the memory access network. A conflict occurs if the address is in the same bank as the address in rank C, B, or A. Conflicts can occur only with scalar or I/Ø references. The scalar instruction senses the conflict condition at issue time + 1 CP. The scalar instruction address enters rank A of the memory access network at issue time + 1 CP. The scalar instruction address enters rank B at issue + 2 CP. The scalar instruction address enters rank C at issue + 3 CP.

Scalar load instruction timing (no conflict):

```
CP n       Issue, reserve register
CP n+1     Address rank A, sense conflict
CP n+2     Address rank B
CP n+3     Address rank C
  .
  .
  .
CP n+10    Clear register reservation
CP n+11    Complete and issue waiting instruction
```

APPENDIX B. WHAT HAPPENS WHEN YOU RUN ON THE CRAY?

You type your LOGON at a terminal; then: (*)

1. The LOGON line goes to the COMBO checker, which verifies it, appends some bits of information and sends it on.

2. The line next arrives at the TMDS concentrator, which notes that it is destined for the CRAY and routes it to the A410.

3. The A410 performs the appropriate protocol and drops the line onto the NSC bus.

4. The A130, which is attached to a CRAY channel, picks up the line from the bus and sends it along the CRAY channel to an LTSS memory buffer.

5. LTSS, which is frequently polling all CRAY channels, notices the activity, sees that this is a LOGON line, and verifies that you are an authorized user.

6. LTSS then prepares an index of private and public disk files to which you have access and associates it with your user number.

7. LTSS returns an appropriate acknowledgment of your LOGON and sends it on the reverse route to your teletype.

The acknowledgment response and all subsequent message lines bypass the COMBO checker. In fact, if the COMBO checker was down at initial LOGON time, the LOGON line would go directly to the TMDS concentrator.

----

* For the MFE network, replace items 1 through 4 above by the following:
M1. The LOGON line goes via a modem and telephone lines to a VADIC modem multiplexor, which sends it on (or it may go directly to step M2).
M2. A PDP-11 concentrator then notes that it is destined for the CRAY and routes the line to a 7600 PPU (12). (In the future, another PDP-11 will be used.)
M3. The PPU performs the necessary protocol and sends the lines to the CRAY-7600 Adaptor.
M4. The adaptor, which is attached to a CRAY Channel, picks up the line and sends it along to a CTSS memory buffer.

Next, you type in an EXECUTE line, say, CLASS / 1 .7, which goes to CRAY LTSS,

1. A search is made of your private file index to determine whether you have a file by the name of CLASS.

2. If not, a search is made of your PUBLIC file index to see if it has a file by that name.

3. If not, the message "NO FILE" is sent to your terminal.

4. When CLASS is found, your PRIORITY is checked (V/TL = .7), and if necessary, changed to conform to the current limits, or, if your account has no time left, changed to S (standby).

5. The job is then assigned to an appropriate loading queue and, when memory space is available, a number of words equal to the load length of this file is brought into memory.

6. When the file is in memory, LTSS performs a sequence of validity checks on the minus words. If any check fails, an appropriate message is returned to your terminal, and execution ceases.

7. If all seems well, the job is placed in an appropriate queue and scheduled for CPU time.

8. When the proper time arrives, LTSS relinquishes control of the CRAY CPU to your program by exchanging from MONITOR to JOB mode, putting the contents of your minus words into the CRAY registers, and requesting the 16-word buffer-load of instructions containing the instruction addressed by your program counter to be fetched to an instruction buffer.

9. Finally, then, the first instruction will be performed and the program counter advanced to the next instruction.

10. In general, your program continues in control of the CPU until it makes a recognized error, gives control back to LTSS, or is interrupted by LTSS. However, while it is in control of the CPU, LTSS may have on-going I/O activity, which will share the use of memory with your program.

All this detail is incorporated in the code CYCLES.

There are essentially five registers to consider, a few flags and a few time positions.

```
                  -------    ------------
                  ! BOO !<---->!     P     !<--!
                  -------    ------------   !
                                 !   !       !
                                 !   -- +1--
                                 !
                                 !
                                 !
---------------                  !
! Instruction !                  !
!             !          -------------   -------------   -------------
!   Buffers   !-------->!   ILATCH   !->!    NIP     !->!    CIP     !
!             !          !            !  !            !  !            !
!             !          -------------   -------------   -------------
!             !                  !                              !
---------------                  !                              ----->
                                 !                                    Execution
                                 !               -------------
                                 --->!    LIP    !------------->
                                     -------------
```

An instruction which issues at cycle x must have entered the CIP at cycle x-1 or before, the NIP at cycle x-2 or before, and the ILATCH at x-3 or before. Some time prior to cycle x-3, the instruction must have been located in one of the four 64-parcel instruction buffers, and before that, it was in memory.

In general, instructions coming from the instruction buffers are able to reach the CIP at a rate of one per cycle; however, when the end of a buffer is reached, delays are encountered in locating the next instruction to be processed. Similarly, whenever Branch instructions cause the orderly flow of sequential instructions to be interrupted, delays are to be expected.

The chart (pages 60-61) illustrates details of the flow of instruction parcels in the CRAY-1. Registers involved in this flow are described in the "Instruction Issue and Control" section of Chapter 3 of the CRAY Hardware Reference Manual.

In general, the P register is incremented by one each time an instruction is issued. If the instruction parcel corresponding to the new P value in sequence is in the current instruction buffer, then that parcel goes to the ILATCH register during the same cycle. If the parcel is not in the current I-buffer, then the ILATCH INVALID flag is set.

If the required parcel is not in any I-buffer, then a memory instruction fetch request (IFR) is issued. Normally, four instruction words (16 parcels) including the required parcel will arrive in the next I-buffer eleven cycles after the IFR. If memory is already busy then the IFR must wait. The other twelve instruction words to fill the I-buffer will be requested in groups of four during the next three cycles. The required parcel reaches ILATCH in the same cycle it reaches the I-buffer. I-buffers are loaded in strict rotation regardless of when the buffer was used last.

If the required parcel is already in a different I-buffer, then CHANGE BUFFER is set and on the following cycle the current I-buffer designator is switched. The correct parcel will reach ILATCH on the following cycle, two cycles delayed. A jump within the current I-buffer takes as long as a jump to a different I-buffer.

An instruction issues from the CIP (current instruction parcel) register. The second parcel of a two-parcel instruction issues from the LIP (lower instruction parcel) register. In the same cycle a new parcel moves into CIP from the NIP (next instruction parcel) register unless blocked by the TPS (two parcel split) flag. The TPS flag is set when ILATCH is invalid and NIP contains the first parcel of a two parcel instruction. (17d)

In the same cycle that a parcel moves from NIP to CIP, a parcel moves from ILATCH to NIP unless blocked by the ILATCH INVALID flag described above. If NIP contained the first parcel of a two parcel instruction, then the parcel in ILATCH goes to LIP instead, and a NOP is placed in NIP.

With these rules we are now ready to use the chart below which illustrates the cycle-by-cycle progress of instruction parcels for the following code sequence:

```
     addr    parcel    CAL mnemonics

      17a    072700    s7   rt
      17b    020100    a1   two
      17c    000002
                      *repeat 1
      17d    031110    a1   a1-1
      20a    030001    a0   a1
      20b    011000    jan  *-2
      20c    000077
      20d    072600    s6   rt
      21a    004000    ex
                      two = 2
```

Assume that completion of an exchange sequence results in setting the P register to 17a in cycle 1.

IFR means "instruction fetch request" issued for these words. x column shows nip entry blocked because invalid data in ilatch. - means invalid or irrelevant data.

cycle	IFR words	words ready	p reg	ilatch	x	nip	(lip)	cip	instr. issued	comments
1	14-17	-	17a	-	x	-	-	-	-	IFR for 14a-17d
2	0- 3	-	17a	-	x	-	-	-	-	(ready in I-buffer
3	4- 7	-	17a	-	x	-	-	-	-	11 cycles after
4	10-13	-	17a	-	x	-	-	-	-	memory request)
5	-	-	17a	-	x	-	-	-	-	waiting
6	-	-	17a	-	x	-	-	-	-	for
7	-	-	17a	-	x	-	-	-	-	instructions
8	-	-	17a	-	x	-	-	-	-	to
9	-	-	17a	-	x	-	-	-	-	arrive
10	-	-	17a	-	x	-	-	-	-	from
11	-	-	17a	-	x	-	-	-	-	memory
12	-	14-17	17a	17a		-	-	-	-	11 cycles after IFR
13	-	0- 3	17b	17b		17a	-	-	-	
14	-	4- 7	17c	17c		17b	-	17a	-	
15	-	10-13	17d	17d		nop	(17c)	17b	17a	s7 = rtc at this cycle
16	20-23	-	20a	-	x	17d	-	nop	17b	IFR for 20a-23d
17	24-27	-	20a	-	x	-	-	17d	nop	a1 now set to 2
18	30-33	-	20a	-	x	-	-	-	17d	a1-1 to address adder
19	34-37	-	20a	-	x	-	-	-	-	
20	-	-	17a	-	x	-	-	-	-	a1 now set to 1
21	-	-	17a	-	x	-	-	-	-	waiting for
22	-	-	17a	-	x	-	-	-	-	instructions
23	-	-	17a	-	x	-	-	-	-	to
24	-	-	17a	-	x	-	-	-	-	arrive
25	-	-	17a	-	x	-	-	-	-	from
26	-	-	17a	-	x	-	-	-	-	memory
27	-	20-23	20a	20a		-	-	-	-	11 cycles after IFR
28	-	24-27	20b	20b		20a	-	-	-	
29	-	30-33	20c	20c		20b	-	20a	-	
30	-	34-37	20d	20d		nop	(20c)	20b	20a	0+a1 to address adder
31	-	-	20d	20d		nop	(20c)	20b	-	
32	-	-	20d	20d		nop	(20c)	20b	-	a0 ready (=1)
33	-	-	20d	20d		nop	(20c)	20b	-	a-branch flags set
34	-	-	17d	-	x	-	-	nop	20b	17d goes to p-counter
35	-	-	17d	-	x	-	-	-	nop	
36	-	-	17d	17d		-	-	-	-	
37	-	-	20a	-	x	17d	-	-	-	change buffer request
38	-	-	20a	-	x	-	-	17d	-	
39	-	-	20a	20a		-	-	-	17d	a1-1 to address adder
40	-	-	20b	20b		20a	-	-	-	

```
  41      -      -     20c    20c      20b     -     20a    -
  42      -      -     20d    20d      nop   (20c)   20b   20a    0+a1 to address adder
  43      -      -     20d    20d      nop   (20c)   20b    -
  44      -      -     20d    20d      nop   (20c)   20b    -     a0 ready (=0)
  45      -      -     20d    20d      nop   (20c)   20b    -     a-branch flags set
  46      -      -     21a    21a      20d     -     nop   20b    (drop through)
  47      -      -     21b    21b      21a     -     20d   nop
  48      -      -     21c    21c      21b     -     21a   20d    s6 = rtc = s7+33
  49      -      -     21d    21d      21c     -     21b   21a    exit
```

```
cycle   notes
  1   words 14-17 are requested from memory.
 12   words 14-17 reach I-buffer 0 and parcel 17a enters ILATCH.
 15   parcel 17a issues fourteen cycles after being requested from memory.
 16   17b issues and parcel 20a (words 20-23) is requested from memory.
      In general, the next buffer is requested when 17b issues from the
      old buffer.  If 20a is not in an I-buffer then it will be ready to
      issue after fourteen more cycles, unless further delayed by memory busy.
 30   parcel 20a issues fourteen cycles after 17b issued and IFR.
 32   register a0=0+a1 is ready.  The result is sent to the A0 branch flag
      setting unit.  This would not delay instructions other than jump
      on A0 instructions.
 33   the A0 branch flags are set.
 34   now the Jump on A0 Non-zero can issue which resets the P register.
      A jump to a parcel already in an I-buffer takes 5 cycles for the target
      parcel to issue.
 37   parcel 20a is requested when 17d leaves ILATCH.  20a is in an I-buffer
      and will be in ILATCH in two cycles.
 39   target parcel 17d issues and 20a reaches ILATCH.
 42   parcel 20a issues as in cycle 30.
 46   JAN issues but this time the P register is not reset and we drop
      through.
 48   the real-time clock reading would be 33 cycles greater than cycle 15.
```

CYCLES' output for this code sequence:

```
  loc   instr              res   operand       w b delay    i     c     o     f     r

00017a 072700              s7    rt              A20000    15    16
00017b 0201 00000002       a1    two                       16    17                 2=a1
00017d 031110              a1    a1-1                      18    20                 1=a1
00020a 030001              a0    a1            11B00204    30    32                 1=a0
00020b 011 0000017d        jan   17d            3 00100    34    39     a    39    48

    jump back to repeat at     17d

00017d 031110              a1    a1-1            a         39    41                 0=a1
00020a 030001              a0    a1            2b00204     42    44                 0=a0
00020b 011 0000017d        jan   17d            3 00100    46    51     a    51    60
00020d 072600              s6    rt                        48    49
00021a 004000              ex                   1 02000    50   100
```

TTY input to CYCLES for this example:

```
cycles tty htty.
p17a c15 72700 20100 2 31110 30001 11000 77
p17a 31110 30001 11000 77 72600 4000 end
```

Summary

Instruction look ahead is effectively three parcels (CIP, NIP, and ILATCH). When instruction 17b of a buffer is issued, the first parcel (20a) of the next I-buffer load is sought. If parcel 20a is already in an I-buffer then it is delayed only 2 cycles; if it is not in a buffer, then it should be ready to issue fourteen cycles after it was requested (ie. after 17b issued). The request is delayed until memory is not busy. After the request is accepted memory is busy for six additional cycles.

There are four exceptional cases to consider:

1. If 17c is a branch instruction, then the instruction fetch request (IFR) is delayed until the jump address is decided. The address is decided in the jump issue cycle except for "J Bjk" in which it is decided two cycles later.

2. If 17c is a scalar load or store which issues immediately, then it gets memory service first and the instruction fetch is delayed four cycles.

3. If 17c is a vector load or store or a block register transfer and it issues immediately, then the instruction fetch is delayed until 17c is done with memory. The delay will be VL+4 for a load and VL+5 for a store.

4. If 17c is a one parcel instruction followed by a two parcel instruction, then if 17c does not issue immediately, it will be held from issue until the second parcel of 17d reaches ILATCH. The hold is caused by the setting of the TPS (two parcel split) flag after 17d reaches NIP.

The following sequences, which differ only by the second instruction issued (at cycle 2 or 1), illustrates this effect:

```
   loc   instr              res   operand       w b delay    i      c      o      f      r

00017a 061106               s1     -s6            A          0      3
00017b 054521               s5     s5<17         1 00020     2      4
00017c 070210               s2     /hs1                      3     17
00017d 1305 00010000        10000b,0   s5       11B00200    15
00020b 064432               s4     s3*fs2                   17     24
00020c 1304 00010001        10001b,0   s4        6 00004    24

   loc   instr              res   operand       w b delay    i      c      o      f      r

00017a 061106               s1     -s6            A          0      3
00017b 042521               s5     <47                       1      2
00017c 070210               s2     /hs1         11 00204    13     27
00017d 1305 00010000        10000b,0   s5         B         14
00020b 064432               s4     s3*fs2       11 00004    27     34
00020c 1304 00010001        10001b,0   s4        6 00004    34
```

In the first case, parcel 17b was delayed one cycle by an S-reg path conflict, so parcel 17c was able to issue immediately and beat the TPS hold. In the second case, parcel 17b had no trunk conflict and issued on cycle 1. Parcel 17c was, as before, ready to issue on cycle 3, but by then the TPS hold was on. Thus, 17c had to wait for 20a to reach ILATCH before the hold was released permitting it to issue.

Work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract number W-7405-Eng-48.

HN/SV